The Paradigm Change from Thin to Rich Clients in Modern Web Applications

An analysis of how the MVC pattern is being shifted to the client side

for the

Bachelor of Science

in Applied Computer Science

at the Baden–Wuerttemberg Cooperative State University Stuttgart

by

Tim von Oldenburg

September 2012

Time of Project	12 Weeks
Student ID, Course	0834311, TIT09AIA
Company	IBM Deutschland MBS GmbH, Herrenberg
Supervisor in the Company	Daniel Suski
Reviewer	Paul Hubert Vossen

Author's declaration

Unless otherwise indicated in the text or references, or acknowledged above, this thesis is entirely the product of my own scholarly work. This thesis has not been submitted either in whole or part, for a degree at this or any other university or institution. This is to certify that the printed version is equivalent to the submitted electronic one.

Stuttgart, September 2012

_____

Tim von Oldenburg

Abstract

The Model–View–Controller design pattern is used in many software architectures to provide a clear application structure. Following the principle of *Separation of Concerns*, it decouples business data from business logic and presentation, and thus improves reusability and maintainability. With the use of MVC in web applications, the challenge emerges to apply the pattern to client–server environments. The result of distributing the pattern components on client and server is a range of different architecture variations.

The applicability of these web architecture variations based on Model–View–Controller is examined in this thesis by means of different application requirements. To demonstrate the shift from thin to rich client web applications, the architecture of *IBM Content Navigator* is presented as an example. The development of a plug–in for IBM Content Navigator further illustrates how rich client MVC applications can effectively be implemented.

Acronyms

AJAX Asynchronous JavaScript and XML

AMD Asynchronous Module Definition

API Application Programming Interface

ASP Active Server Pages

CGI Common Gateway Interface

CRUD Create, Read, Update, Delete

CSS Cascading Stylesheets

DOM Document Object Model

EAR Enterprise Archive

ECM Enterprise Content Management

EJB Enterprise JavaBeans

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

I/O Input/Output

IBM International Business Machines Corporation

ICA IBM Content Analytics

ICN Content Navigator

ICN IBM Content Navigator

IDX IBM Dojo Extensions

IETF Internet Engineering Taskforce

IIS Internet Information Services

JAR Java Archive

Java EE Java Platform, Enterprise Edition

JRE Java Runtime Environment

JSON JavaScript Object Notation

JSP JavaServer Pages

MVC Model–View–Controller

MVP Model–View–Presenter

MVVM Model–View–ViewModel

OOP Object-Oriented Programming

PHP PHP Hypertext Preprocessor

REST Representational State Transfer

SaaS Software–as–a–Service

SCCM SmartCloud Content Management

SQL Structured Query Language

SSO Single Sign-On

TCP Transmission Control Protocol

UI User Interface

URL Uniform Resource Locator

W3C World Wide Web Consortium

WAS WebSphere Application Server

XML Extensible Markup Language

Contents

1 Introduction

Consumer applications have existed on the web for more than a decade now. Amongst others, e-mail, time management, social collaboration and file sharing applications can all be accessed using a web browser. For consumers, the main advantages are the independence of special software and the ability to access their data from nearly any computer, but web applications are also an interesting approach for enterprise solutions: In opposition to desktop applications, web applications do not need to be installed on end-user machines. This advantage is complemented by the fact that the roll-out of updates can easily be performed on the server-side only.

Until recently, browsers were not powerful enough for desktop-level, complex end-user clients. But as rendering techniques and JavaScript engines get faster, the possibilities of grow. In combination with JavaScript frameworks, web browsers become a runtime environment, powerful enough to build applications that can replace traditional end-user clients.

The Model–View–Controller pattern is a commonly used architectural design pattern. Being based on the principle of *Separation of Concerns*, it helps to structure an application by dividing it into three different parts: the Model, which holds business data, the View, which displays these data, and the Controller, which is responsible for user interaction.

MVC was created in the 1970s, long before the world wide web became popular, and has been used in many desktop applications since. But applying it to the domain of web applications introduces a challenge: the web is a distributed computing environment. While it is rather simple to connect MVC components inside a monolithic desktop application, distributing them across different tiers — at least *client* and *server* — opens up possibilities for different implementations. With respect to three defined criteria, this thesis investigates and evaluates different web application architectures based on the Model-View-Controller pattern.

1.1 Motivation and Scope

In the following section, *IBM Content Navigator*, one of IBM's most recent web clients, is introduced. By connecting it to Enterprise Content Management and SmartCloud Content Management, the business problem that leads to the goals of this thesis is explained.

Enterprise Content Management and IBM Content Navigator

According to Wikipedia (n.d.*a*), "Enterprise Content Management (ECM) is a formalized means of organizing and storing an organization's documents, and other content, that relate to the organization's processes. The term encompasses strategies, methods, and tools used throughout the lifecycle of the content." Contracts, invoices, orders or insurance policies are all examples for such content. IBM offers different solutions for ECM, three of them being to manage digital documents: *FileNet P8 (P8)*, *ContentManager (CM)* and *ContentManager OnDemand (OD)*. All three are complemented by web clients used to access the repositories (e.g. to upload, download, or move documents): *Workplace XT* for FileNet and *WEBi* for CM and OD. They are usually delivered with every solution that also includes their respective repository software.

Amongst other clients of IBM ECM products, Workplace XT and WEBi are going to be replaced by IBM Content Navigator (ICN)[1]. This new web application allows access to FileNet, ContentManager and ContentManager OnDemand repositories through a single User Interface. IBM Content Navigator is a Rich Client based on the Model–View–Controller pattern. It makes use of *The Dojo Toolkit*, which is an open source JavaScript framework to build rich, client-side web experiences based on widgets and the *AJAX* technique. Dojo is the standard framework used by most of the IBM web clients. ICN follows IBM OneUI, which is an initiative within IBM to unify the user experience[2] in products' web clients by providing guidelines. Content Navigator also makes use of IBM internal extensions to Dojo, called the *IBM Dojo Extensions*, which primarily enhance existing Dojo widgets to follow the OneUI standards.

[1] See http://www-01.ibm.com/software/ecm/experience.html
[2] The look and feel of a User Interface.

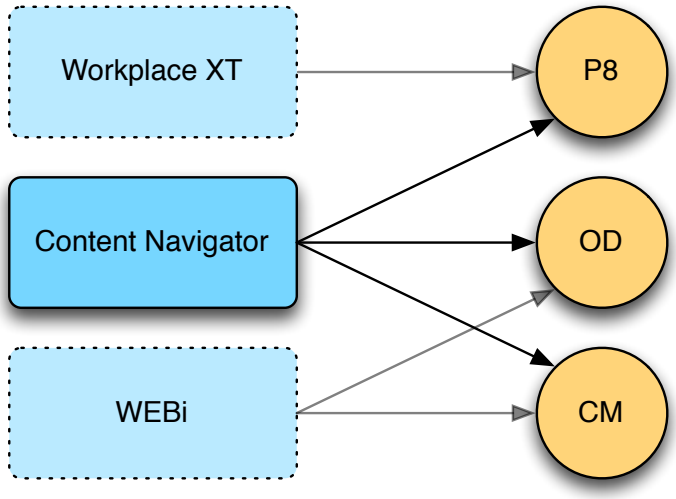

Figure 1.1: IBM Content Navigator replaces WEBi and Workplace XT

Due to its plug-in based architecture, IBM Content Navigator can not only be used as a distinct application, but also as a framework which can be used to build arbitrary web applications. Although originally built with the access to ECM content repositories as a core feature, web applications developed on top of ICN can serve any purpose.

Content Navigator makes integration of different applications possible through two features:

- A plug–in system that allows developers to create extensions for every purpose and on every layer with both server-side and client-side features
- A client-oriented Model–View–Controller (MVC) architecture that provides the structure for the rich, browser-based application

SmartCloud Content Management

IBM SmartCloud Content Management (SCCM) is a Software–as–a–Service (SaaS) offering for document and email compliance archiving. It is part of IBM's ECM portfolio and integrates different IBM products to deliver an out-of-the-box archiving solution for enterprises. One of these integrated products is the document management system *FileNet P8* and its web client, *Workplace XT*. In addition to Workplace XT, SCCM also provides a browser-based administration interface.

As Workplace XT is going to be replaced by ICN, it is planned to port the SCCM administration interface to the ICN platform for the next release, too. This integration will provide customers with a unified User Interface for the different web clients delivered through SCCM.

Goals and Scope

As a first goal, this thesis demonstrates how the MVC pattern can best be implemented in web applications. Based on three defined criteria, different MVC-based web application architectures are evaluated to investigate the shift to rich, JavaScript-based clients.

The second goal of this thesis is to investigate and present the part of the IBM Content Navigator architecture related to the MVC pattern. In conjunction with the thesis of Robert Metzger, a working prototype of an ICN plug-in was developed to approach a problem being faced with the SCCM cloud offering. The plug-in demonstrates how the MVC pattern can be implemented in practice and should help the SCCM development team to get up to speed with porting the SCCM administration interface to IBM Content Navigator.

It is not in the scope of this thesis to create a new way to implement MVC in web applications. It is also neither a goal to find and list all possible implementations of MVC architecture on the web, nor to find the single best way to implement it. Instead, four MVC architectures and their applicability are compared to identify the one that best fulfills an application's requirements.

1.2 Document Structure

Chapter One, "Introduction" explains the motivation behind this thesis. It familiarizes the reader with the business background, defines goals and outlines the document structure.

Chapter Two, "The Model–View–Controller Pattern" introduces MVC, its history and origins in Smalltalk-80, its structure and the tasks of the different components. It also presents two variations of MVC — MVP and MVVM — which are commonly used in web applications.

Chapter Three, "Web Application Architecture" describes technologies and techniques relevant to web application architecture and explains the terms *Thin Client* and *Rich Client*.

Chapter Four, "Applicability of Different MVC Architectures for Web Applications" gives examples of problems that emerge with web applications and defines these problems as criteria. After that, various possibilities to implement MVC in client–server architectures are discussed based on the previously defined criteria.

Chapter Five, "MVC in IBM Content Navigator" presents IBM Content Navigator and examines its plug–in system and architecture. Hereafter, the design of a log analysis plug–in developed in the course of this thesis is discussed further.

*Chapter Six, "**??**"* summarizes the findings and gives an outlook on web application architecture with respect to current developments and evolving standards.

2 The Model–View–Controller Pattern

This chapter discusses the Model–View–Controller pattern. It introduces design patterns in general, the classical Model–View–Controller architecture and different interpretations and variations of it.

2.1 Design Patterns in Software Engineering

The principle of using design patterns was first introduced by Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King & Angel (1977). The architects describe how a community can shape towns, neighbourhoods and buildings with simple tools. To communicate architectural knowledge in an effective way to the readers, he introduces *design patterns*:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core to the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." (Alexander et al. 1977, p. 10)

This shows the main aspects of a design pattern:

1. A pattern provides a solution to a specific, frequently occurring problem
2. The pattern's solution is generic (only "the core to the solution", but no implementation details)
3. The main advantage of a pattern is reusability

Design patterns are a method of communicating solutions that have proven to solve a specific problem. Alexander's definition is open enough to be applied to nearly any field that deals with design problems, for example graphic design, architecture, interaction design, and software design. Most of these areas have repositories of design patterns. For architecture, there is the aforementioned book by Christopher Alexander, interaction design patterns can be found at

the *Yahoo! Design Pattern Library*[1], and software design patterns are listed in the *Portland Pattern Repository*[2].

Gamma, Helm, Johnson & Vlissides (1995) describe design patterns that are used in object-oriented software development. There are design patterns for other programming paradigms, but most of the more popular ones are created for object-orientation, especially as they leverage the given reusability that OOP provides through classes, interfaces, or .

Design Patterns simplify software development. If a pattern used in a software is well documented and understood, it is easier to maintain the software, to reuse code, and easier to communicate software architecture to new developers.

Gamma et al. classify the described design patterns into three categories, depending on their purpose.

Creational Patterns provide a way to create objects under certain circumstances (also known as *object instantiation*). A popular example of creational patterns is the *Singleton*, which ensures that only one instance of an object exists at a given time.

Structural Patterns describe the composition (or *structure*) of different objects or classes. They offer additional ways of composing as those already given through object-orientation (such as composition, aggregation or association). An example for a structural pattern is the *Proxy*, which simulates the existence of a remote or otherwise unaccessible object by copying its interface and forwarding method calls. Amongst other purposes, proxies are used to simplify remote access or to realize access control.

Behavioral Patterns are used to define the interaction of objects or classes, the flow of control or the distribution of responsibility. The *Chain of Responsibility*, for example, is a sequence of interconnected objects that handle specific input. Depending on the type and content of this input, an element in the Chain of Responsibility may or may not process the input and pass it on to its successor.

According to Gamma et al. (1995), design patterns exist on different levels of abstraction. The patterns described in "Design Patterns" are mostly scoped on classes and objects. On a higher level of abstraction, there are **Architectural Patterns**[3], one of them being *Model–View–Controller*. Architectural patterns are not described by Gamma et al., but MVC in particular is mentioned as

[1] See http://developer.yahoo.com/ypatterns/
[2] See http://c2.com/ppr/
[3] There are patterns of software architecture, not to be confused with the architecture patterns that Alexander et al. (1977) described.

an illustrative example in the introductory chapter (1995, p. 4–6). A collection of architectural patterns is listed by Fowler (2002).

2.2 History and Interpretation of MVC

This section outlines the structure and control flow of Model–View–Controller. It explains the origins of MVC, describes the different components and discusses what leads to the different interpretations and variations of MVC.

2.2.1 Origins of MVC in Smalltalk-80

Model–View–Controller was first introduced by Trygve Reenskaug while he was working with the group at Xerox PARC from 1978 to 1979. He designed several versions that included the components "Editor" and "Thing"[1] before he settled on the MVC terminology. Reenskaug defined the MVC pattern for the use in applications with a Graphical User Interface (GUI). His initial idea was to map the user's mental model of the data to the computational representation of the data. For this to achieve, he wanted to abstract the data, using different components and hiding the real Model from the user — a principle that is called *Separated Presentation*. Reenskaug summarized his work on MVC in "The Model–View–Controller (MVC). Its Past and Present" (2003).

According to Osmani (2012, p. 99ff.), MVC takes the separation one step further and makes a clear division between the user interface and the application logic. This concept is now called — more generally — *Separation of Concerns*. Before Smalltalk-80 MVC, Graphical User Interfaces were designed as one single module of code. This module handled the data, the presentation and the user interaction. While this worked well for tiny applications, it was hard to maintain for regular or large applications. Separating the presentation, user interaction (logic) and data allows to reuse these components.

[1] Editor and Thing are components that were part of very early versions of MVC. They are not further discussed here.

2.2.2 MVC Components

Model, View and Controller are the components of a MVC architecture. Due to the high interpretability of this pattern, a common definition of the components and the pattern structure could not be found during research for this thesis. Thus, the definitions below are based on the different sources of Trygve Reenskaug, Gamma et al. and Martin Fowler.

The Model

MVC-based applications are data-centric, which means they are based on *knowledge* that is organized in one or more *Models*. A Model is the programmatic representation of these data, either the whole set or only a part.

Neither the characteristics of the data nor the way they are stored within the Model is relevant in this context. Data may be

- *structured*, such as a numerical table or a filled-out form
- *semi-structured*, such as a XML or Microsoft Word document
- *unstructured*, such as the body of an e-mail (free-form text).

A Model may store a particular element of its data in any way, depending on the data characteristics, the programming paradigm used and the requirements of the application. Possible storage models include, but are not restricted to

- *Objects* as used in object-oriented programming languages, such as Java objects or JavaScript objects
- *Arrays* and *Lists* as used in most programming languages
- *Plain binary or textual streams* like files. A file system along with I/O functions can act as a Model.

Reenskaug described the original MVC assuming Models would be Smalltalk objects, but with the spread of other programming languages and databases, every storage model can be imagined.

Often, Models contain more than just the plain data they embody. As is stated in Reenskaug (1979*b*), a Model is "represented in the computer as a collection of data together with the methods necessary to process these data." This means, a Model can also contain functions, such as

to *create*, *read*, *update* and *delete* data (also knows as *CRUD*). Auxiliary structures, such as hashes and indices, as well as functions add data-related business logic to the Model. They can simplify the handling of data extremely, for example through input validation, data conversion, sorting capabilities and faster access.

A concept found in many collections of data, such as Java objects or relational databases, is the dependance of data on other data. A Java object, for example, can have the reference to a different object as one of its attribute values. In relational databases, a can refer to another tuple, which is part of either the same or a different , using a foreign key. Those references have to be considered when designing Models.

In many cases, the data source is not the actual Model, depending on the point of view. If a Java application implementing Model–View–Controller needs to access data in a relational database, this is most probably implemented using an object-relational wrapper. From the point of view of the Java application, the wrapper classes are Models, while from an architectural point of view, the database (or the database tier) represents the Models. Such ambiguities lead to the need of good communication in the design and development teams, and they are one reason for the various interpretations of the pattern.

To clarify what the "Model" means in a given context — for example, in one development team — Reenskaug (1979*b*) proposes that all Models should have the same level of abstraction. That means, "Model" should never describe a database and, at the same time, an object holding data from this database, which would be more concrete than the database itself and thus on a different level of abstraction.

The View

As MVC is a pattern for applications with a user interface, the View is responsible for (visual) data presentation. It can reach from a simple, textual representation to tables, charts and diagrams or even more complex output. A View is always connected to a Model and is used to present the Model's data. A View is usually related to one Model only, whereas a Model can have multiple Views.

The output medium according to Reenskaug (1979*b*) is either the screen or hardcopy (print), but nowadays other output media, such as audio (for example a speech synthesizer) or haptics (for example a *braille* device used by blind people to interact with a computer) can be applied to the View concept. However, in software development, the screen is still the main output device,

and is focused on in this definition; therefore, the terms “present”, “show” and “display” are used synonymously.

Usually, not all data of a Model are presented at once. There are two ways of limiting the presented data, both of which can be compared to relational algebra for illustration. On the one hand, a View can display only several chosen facets of the Model. This can be compared to a *projection* in relational algebra. On the other hand, the set of data can be limited, so that only items that match certain criteria or not more than a specific number of items are displayed. This can be compared to a *selection* in relational algebra.

As the View is an interface to the user, the functionality described above is needed in order to let the user focus on the important part of the data. Scrolling, pagination and lazy loading are all examples for features that support this functionality of the View component. Reenskaug (1979*a*) calls the View a “presentation filter”.

To fulfill the task of presenting a Model’s data, a View has to know about the Model. (In contrast, the Model is totally independent of the View.) This knowledge is needed for two reasons: On the one hand, it needs to have access to the actual data to be able to display them. On the other hand, a View may need to know metadata to display the actual data properly.

To illustrate the latter, we can imagine a table (View) that displays data from a relational database (Model). Whereas some columns show text or numbers, another one displays dates which have to be formatted according to the user’s local settings. This cannot be the Model’s task; it is aware of the fact that this piece of data is a date, but it does not know how to format a date properly. This would be the task of the presentation component, which in Model–View–Controller is the View.

The data a View displays are held in the View’s private buffer. When the Model changes, the View has to be informed about that to update these data. To fulfill this, the View is connected to the Model via an Observer pattern as described by Gamma et al. (1995) (see Section 2.2.3 for more information on the Observer).

The Controller

The exact role of the Controller is probably the most discussed part of Model–View–Controller. The resulting variations, such as Model–View–ViewModel or Model–View–Presenter all kept the Model and View, but redefined what used to be the Controller. To clearly draw a picture of the Controller’s task, Reenskaug’s definition is taken into account:

> "A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and pass these messages on to one or more of the views." (Reenskaug 1979*a*)

In other words, the Controller is the part of an application that implements the overall application flow and the user interaction. As already mentioned, Model–View–Controller is an architectural pattern for applications with a Graphical User Interface. The Controller processes user input, for example such input that is used to manipulate the Model.

Some interpretations add the term of an *Application Controller* to pay respect to the fact that there are Controllers on different layers of the application, as described by Fowler (2002, pp. 379–386). A View-bound Controller may react when the View is clicked, but it can only know that from a higher–level Application Controller that tracks mouse movements and events. In the case of web applications, this is the browser (or the JavaScript engine, respectively). According to Fowler (2006*a*), there is "not just one view and controller, you have a view-controller pair for each element of the screen, each of the controls and the screen as a whole."

The term *Front Controller* describes an object that manages the application state and navigation by providing a single entry point to the application[1]. In the area of web pages, a Front Controller is a "A controller that handles all requests for a Web site." according to Fowler (2002, p. 344). One example for such a Controller is the Servlet employed in *Model 2*, a Java EE MVC architecture using JSPs, EJBs and Servlets (see Section 3.2.2).

2.2.3 Model, View and Controller in Interaction

The separation of data, presentation and interaction is a powerful way to structure an application. The goal of MVC is reusability and maintainability, but the three components still have to work together to make sense. The introductory chapter of "Design Patterns" describes the different design patterns used within MVC (Gamma et al. 1995, pp. 4–6).

[1] The Front Controller is called *Router* by Osmani (2012).

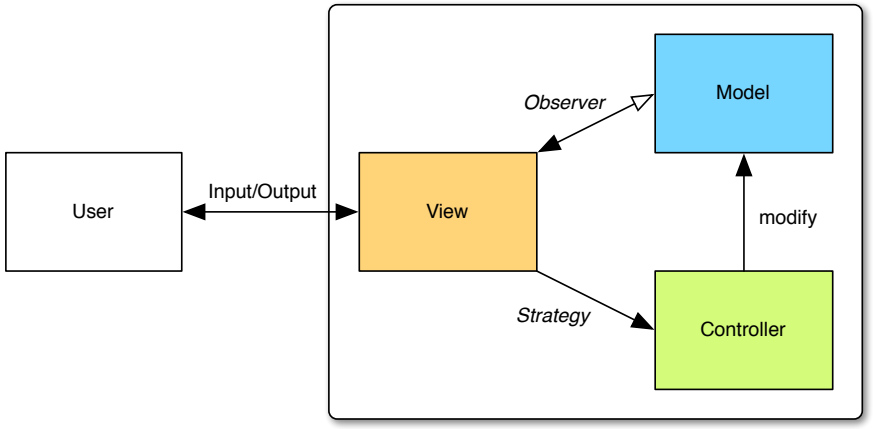

Figure 2.1: Structure of Model–View–Controller

Model and View

According to Gamma et al. (1995, p. 4), "MVC decouples views and models by establishing a subscribe/notify protocol between them." In other words, the View "subscribes" to the Model, and the Model then notifies the View whenever the Model's data change. Eventually, the View can update itself to reflect those changes. The *subscribe/notify* solution described by the *Observer* pattern in Gamma et al. (1995, pp. 293–303) is also called "Publish–Subscribe" and allows multiple Views to observe a single Model (see Figure 2.2).

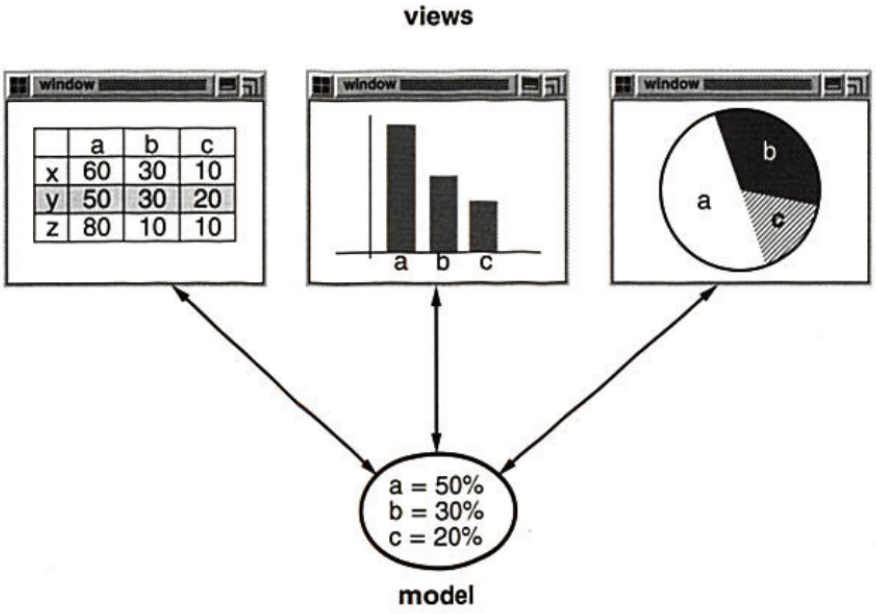

Figure 2.2: Multiple Views connected to a single Model

Source: Gamma et al. (1995, p. 5)

According to Osmani (2012, p. 108), the "observer nature of this relationship is what facilitates multiple views being attached to the same model." This means that Model and View are completely decoupled. The Model does not know about the View, it only notifies all its subscribers when it changes. A subscriber can be any arbitrary object, which only needs to implement a special function to receive the notification (in Java, this is typically realized using an *interface*). Thus, a subscriber does not even have to be a View, which means that the Model is completely isolated from the User Interface (UI) part of the application (if there is any).

View and Controller

The relationship between View and Controller is defined by a Strategy pattern, or in other words: the View uses the Controller as a Strategy object. A Strategy object is an object representing an algorithm. If an interaction is triggered on the View, for example a key press, mouse movement or mouse click, the View recognizes it and invokes the Controller. The Controller then decides on how to react to the according event.

A good example is *scrolling* in a scrollable list. This list (the View) is constrained in its height, so it can only display a part of the list items. List items are stored in the Model. As soon as the user

initiates scrolling, for example by moving the mouse wheel or moving the scoll bar, the View calls the Controller. The Controller recognizes the event as scrolling and calls back the View to update itself. Depending on how far the user scrolled, other list items are shown in the list. This course of events is also illustrated in Figure 2.4.

On his website, Reenskaug describes the pair of View and Controller as the *Tool* (see Figure 2.3). But, as the Gang of Four writes in "Design Patterns: Elements of Reusable Object-Oriented Software", it is important to keep this coupling loose. This allows a software developer to easily replace a View's Controller, even at runtime, to change its interactive behaviour. The example from Gamma et al. (1995, p. 6) proposes that, to disable interaction on a View, one could assign to it a Controller that ignores any interaction.

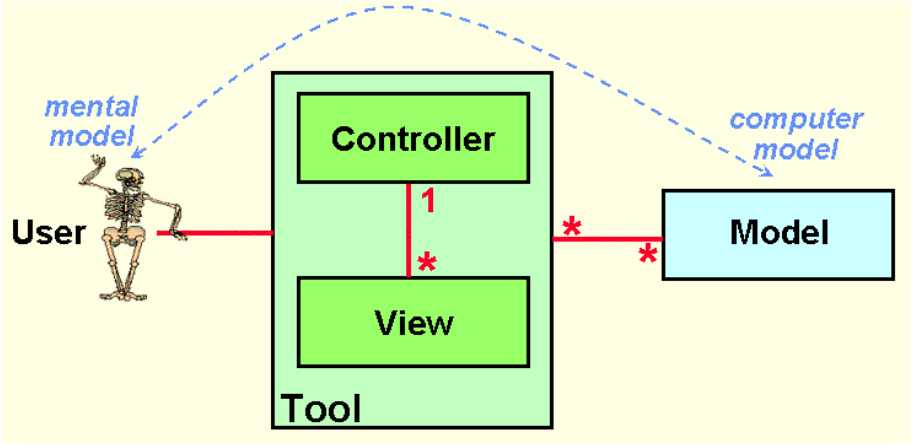

Figure 2.3: The Tool contains Controller and View(s)

Source: Reenskaug (n.d.)

Controller and Model

As described above, one task of the Controller is to react to the user's input. When the user's interaction is targeted at manipulating the domain data, for example by changing the value of a table cell, the Controller's task is to update the Model accordingly.

As mentioned by Osmani (2012, pp. 105, 108) and Reenskaug (1979*a*), the Controller updates the View after the Model changes, but this is not necessary in implementations where the View observes the Model, as is stated in Fowler (2006*a*): "When the model changes, the views react." The sequence diagram of *Figure 6: Changing the actual value for MVC.* by Fowler (2006*a*) emphasizes that (see Appendix). Subsequently, Addy Osmani clarified that it is "absolutely correct that

the Observer relationship should be responsible for updating the View. The Controller should in no way be changing the View directly."[1]

As it can be seen, the role of the Controller is subject to very different interpretations.

Flow of Control

The View and Controller are defined as two very separate objects, the View being responsible for presentation (output), the Controller being responsible for user action (input). It is a matter of definition if a user interaction is performed *on the Controller*, or if it is performed *on the View* and the View then uses the Controller to react to it (*Strategy* pattern). The first variant is illustrated by Fowler (2006*a*) in the sequence diagram of *Figure 6: Changing the actual value for MVC* (see Appendix) as well as by Steele (2004). Some sources, however, assume the latter case (the *Strategy* pattern), as it is described in Gamma et al. (1995, p. 6). In this thesis, both variants are taken into account, depending on which one makes more sense in a given situation.

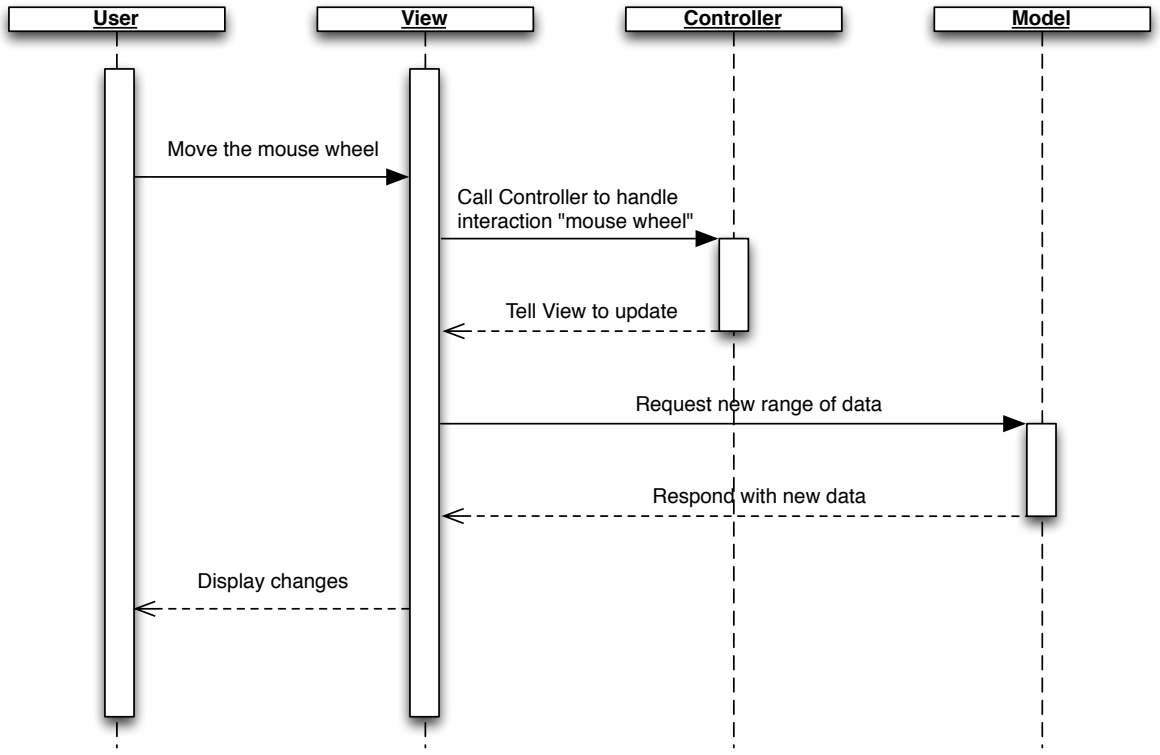

Figure 2.4: Sequence diagram of a User Interaction with Model–View–Controller

[1] Personal conversation with the author, 07/23/2012

The sequence diagram in Figure 2.4 illustrates the user scrolling over a list of items.

2.3 Pattern Variations

The following section describes variations of MVC that can be found in modern software architecture, especially in web applications and the according frameworks. Those variations are an answer to the different interpretations of Model–View–Controller, primarily regarding the responsibilities of the components. MVC and its variations are often called the *MV* family* of patterns (Osmani 2012).

2.3.1 Model–View–Presenter

The first variation of MVC to be discussed is Model–View–Presenter (MVP). It was created in the early 1990s to improve presentation logic for the Taligent operating system and has been adapted in a slightly modified form for the use in web applications by the *Backbone.js* framework[1].

The Model in MVP behaves similar to the one of MVC: it contains and handles domain data. The difference to MVC is in the View and the *Presenter* which replaces the Controller.

The MVP View is called a *Passive View*, because it contains as few logic as possible (thus being "dumb"), as described by Fowler (2006*b*).

The Presenter, however, has additional responsibilities compared to the Controller. In Smalltalk-80 MVC, the View gets the data to display directly from the Model. In MVP, the Presenter separates this connection, acting as a mediator between the View and the Model (Osmani 2012, p. 109). It decouples Model and View completely by breaking up the Observer connection. The original Controller tasks of manipulating the Model and processing user input are in the responsibility of the Presenter, too.

There are two exceptional advantages in combining the Presenter as a mediator with the Passive View, according to Osmani (2012) and Fowler (2006*b*):

- It allows developers to rapidly prototype user interfaces. Most of the presentation and business logic (which is not relevant in the context of prototyping) is in the Presenter, whereas the View is the component being prototyped.

[1] See http://backbonejs.org/

- The thin presentation layer embodied by the Passive View makes automated testing possible, which in other cases is hard to do with a user interface.

Due to the Passive View, View and Presenter are completely decoupled in MVP. There is no data-binding of the View to the Presenter. This means, it is up to the Presenter to decide how to display data in the View, whereas in other implementations (MVC, MVVM) the View contains the presentation logic.

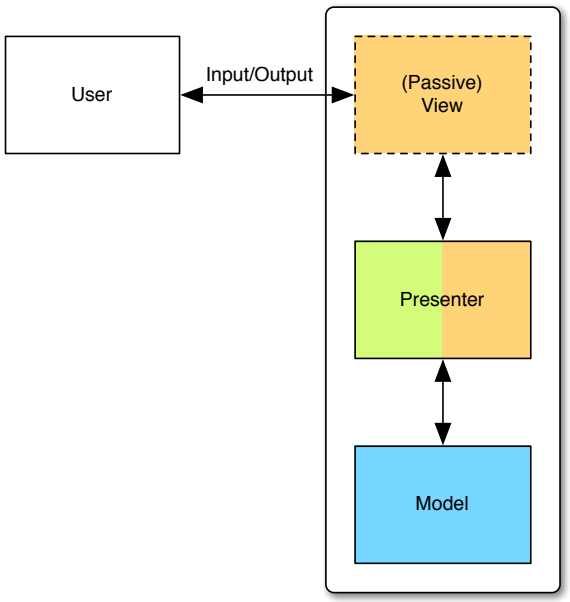

Figure 2.5: Structure of Model–View–Presenter

2.3.2 Model–View–ViewModel

A second variation of MVC widely used in web applications is Model–View–ViewModel (MVVM). It was first introduced with Microsoft Silverlight[1] in 2007. Fowler (2002) describes a similar pattern, called "Presentation Model". MVVM is based on MVC and MVP and includes concepts of both.

Just as in MVC and MVP, the Model contains domain data and the according CRUD functinality.

In opposition to MVP, the MVVM View is not passive. It contains presentation logic and decides if and when to update itself. In MVP, this task is done by the Presenter. The Presenter "pushes"

[1] Silverlight is a browser extension to include rich media, similar to Adobe Flash.

changes of the Model to the View, whereas the ViewModel only "notifies" the View of those changes; it is up to the View to request the changed data and format them to its purpose.

The term "ViewModel" is caused by this component acting like a View-specific Model. Just like the Presenter in MVP does, it decouples the View from the Model and acts as a mediator between them. The difference to a Presenter is, that the ViewModel handles data modelling rather than presentation. It listens for changes in both the Model and the View and synchronizes them, doing only basic data conversions[1].

ViewModel and View are connected through bidirectional data bindings, which act as the single interface between these two components. In its relationship to the View, the ViewModel can be compared to the MVC Model, which also notifies the View of changes, but does not change the View directly.

From a high level point of view, the structures of MVVM and MVP are the same. The differences are in the responsibilities and the coupling of the components; Figure 2.5 and Figure 2.6 illustrate this using colors representing the responsibilities of the MVC components, which are differently distributed in MVP and MVVM.

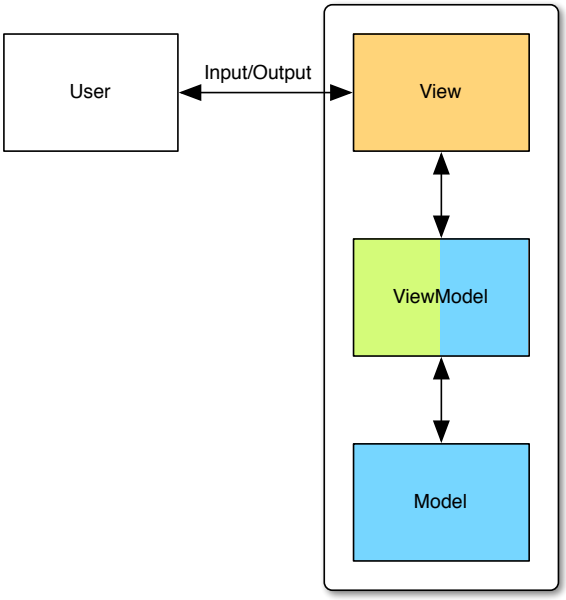

Figure 2.6: Structure of Model–View–ViewModel

[1] For example, the conversion from a UNIX timestamp ("1342177280") to a localized timestamp ("July 13, 2012 11:01:20").

2.3.3 Comparison of MVC, MVP and MVVM

The following example illustrates the differences between MVC, MVVM and MVP: A part of an application displays a list of tasks. In the Model, they are organized as an array of task objects, each of which has different attributes, such as the task name, the due date and the state ("done" or "not done"). There are a hundred tasks, but the UI has to display only the first ten tasks.

In MVC, the View fetches the data it needs (the first ten items) directly from the Model. As the View already is a list presentation, it only needs to add the tasks as list items. The Controller is not involved in this process.

In MVP, the Presenter fetches the items it needs (the first ten) from the Model. It then formats them as a list with every list item having a check box next to it. This list gets forwarded to the View and is displayed there.

In MVVM, the View tells the ViewModel that it needs the first ten task items. The ViewModel then fetches those items from the Model and send them to the View. The View formats them as a list.

Basically, all three patterns fulfill the same tasks. As mentioned before, the difference is in the responsibilities of the various components.

The MVVM View contains more presentation logic than the MVP one, its structure is given and it prepares the data for presentation itself (except for logical conversions). In MVP, this is the Presenter's task, while the View keeps passive.

Both MVP and MVVM decouple the Model from the View, in opposition to MVC. The direct access from the View to the whole Model in MVC is considered bad by Osmani (2012, p. 123), as it can "have security and performance costs". In other words, access to the Model is neither controlled nor cached. The missing ability to cache data in MVC may be a downside compared to MVP and MVVM, but on the other hand, there is less processing of data between View and Model. When data conversions in Presenter or ViewModel are complex, this can be an advantage for MVC.

It is a matter of preference and use case to choose the architectural pattern that suits an application best. In the following chapters, especially in Chapter 4, MVC is assumed to be the pattern used, but the components marked as Model, View and Controller do not have to be distinct objects — it is more important to understand that they fulfill the tasks and have the responsibilities of the respective MVC components, as described earlier in this chapter.

3 Web Application Architecture

This chapter describes the architecture of web applications. After an introduction to different web and web-related technologies and their roles, the differences between *thin client* and *rich client* web applications are discussed.

3.1 Supporting Technologies

The following section gives an overview of different technologies that are used in web applications (and other browser-based applications) nowadays. This list does not claim to be complete, but it covers those technologies important for the discussion.

3.1.1 Server Side

Hypertext Transfer Protocol (HTTP) is the application layer network protocol of the web. HTTP is a textual, stateless protocol using TCP. In a web environment, a web server — such as Apache, Lighttpd, or Microsoft Internet Information Services (IIS) — acts as the HTTP server, whereas a web browser usually acts as the HTTP client. Not only web pages are transferred using HTTP, but also resources like Cascading Stylesheets (CSS), scripts (e.g. JavaScript files), Extensible Markup Language (XML) data and images. HTTP is an open standard developed by the Internet Engineering Taskforce (IETF) and the World Wide Web Consortium (W3C).

HTML Preprocessors are compilers or interpreters that generate Hypertext Markup Language (HTML). Some programming languages are especially built for this purpose, but in general, every programming language and compiler can be used to generate HTML. Some of these preprocessors, such as PHP, Perl and Python, are invoked through the Common Gateway Interface (CGI) of web servers. Others are used with application servers instead,

for example Active Server Pages (ASP), ASP.NET, and JavaServer Pages (JSP). In the area of enterprise applications, JSP has become the established standard.

HTML preprocessors allow the dynamic generation of HTML, XML and other resources through the use of — mostly imperative — programming languages. The power of functions, variables and Application Programming Interfaces (APIs) can be used, for example to query a database and process the received data. This allows to overcome the statelessness of HTTP.

Application Servers provide commonly used APIs and frameworks to applications, enabling them to use features such as security, load balancing, database connections or MVC. Application servers are often used for enterprise applications, as they can shorten development time significantly. The most popular examples, such as Red Hat JBoss, Apache Tomcat, or IBM WebSphere Application Server (WAS), can deploy Java and Java EE applications, but there also are application servers for other programming languages, such as Zope for Python and Zend Server for PHP Hypertext Preprocessor (PHP). Usually, application servers include an HTTP server for the communication with web browsers, but this task can also be handled by a third-party web server.

Java Platform, Enterprise Edition (Java EE) application servers provide a Java Runtime Environment (JRE) and consist of multiple components, including the following:

- The *Servlet container*, also called *web container*, manages *Servlets* and maps them to a Uniform Resource Locator (URL). A Servlet is a Java class that serves a client request with a special response, most often via HTTP. In other words, the Servlet container is the application server's interface to the web browser.
- The *JSP container* translates JSP files into Servlets at runtime. JSP features a templating system for HTML that allows developers to embed Java code into HTML.
- The *EJB container* manages *Enterprise JavaBeans (EJB)*, which are components that encapsulate business logic in Java EE applications. EJBs run back-end code and execute tasks like database access, messaging and web service invocation.

There are more components in the Java EE specification, but they are not relevant in this context.

3.1.2 Client Side

HTML is a markup language to represent hypertext[1]. It was created by Tim Berners-Lee, who also developed the first web browser. HTML consists of elements described by *tags* which are enclosed by angle brackets. Tags often give semantics to elements, such as headings, emphasized text, lists or tables. Web browsers build a Document Object Model (DOM) out of HTML code and display it as a web page.

CSS3 CSS is a W3C standardized language to describe the look ("style") of markup elements. It can be used, for example, to format and style an HTML document. The third generation of CSS adds more possibilities for designing user interfaces for web applications. In the past, visual enhancements such as shadows, gradients or non-standard fonts had to be provided by image files, and animations had to be programmed using JavaScript. CSS3 is capable of doing most of that while using less bandwidth and ensuring standards conformity, thus enhancing the performance and usability of websites. Some modern browsers even use hardware acceleration for CSS3 animations and effects. CSS3 makes complex user interface design possible on the web while separating the presentation from the content.

In combination with CSS preprocessors[2], such as SASS, LESS, or Stylus, development time can be reduced significantly.

JavaScript is, according to Flanagan (2006, p. 1), "the programming language of the web." It is a weakly typed multi-paradigm language (functional and object-oriented), created 1995 by Brendan Eich at Netscape (Eich 2008). It is standardized as *ECMAScript*, the latest version being ECMAScript 5.

Most modern web browsers, such as Mozilla Firefox, Google Chrome, Opera or Microsoft Internet Explorer 9, provide a JavaScript engine and thus are able to execute JavaScript. Although they are not part of the language itself, APIs are provided by browsers to manipulate the DOM and use other browser functions, such as the browsing history. An event–callback system offered by the browser API allows developers to build interactive web

[1] Hypertext resembles the thinking of human beings, in opposition to sequential text. Hypertext contains semantics and links to other texts, and is therefore non-linear.

[2] CSS preprocessors allow the creation of CSS from more versatile and powerful languages that are especially designed for this reason. Those languages, such as SASS, LESS, or Stylus, provide advanced features known from imperative languages, like variables and functions.

sites, in opposition to static ones based solely on HTML. Different libraries and frameworks, such as Dojo, jQuery or Prototype, exist to simplify web application development with JavaScript.

There are also server-side implementations — for example *Node.js*, which is based on Google's V8 JavaScript engine — but they are not relevant in this context.

Asynchronous JavaScript and XML (AJAX) is not a technology, but a method to allow dynamic loading of content, even after the actual web page has completely loaded. Using `XmlHttpRequest`, an object provided by the browser, HTTP requests can be sent from within JavaScript. The response, once returned, can then be processed in the background. The request is asynchronous, which means that the web page is not blocked while waiting for the response; it continues reacting to user input. This is especially important if the web page should feel like a desktop application.

The method is called Asynchronous JavaScript and XML, because the response format often is XML. However, this does not need to be the case. The response can be of any kind, for example an image, HTML or JavaScript Object Notation (JSON) data. JSON is a subset of the JavaScript language, which means it can directly be included into the running script. Literals, objects and arrays can all be serialized as JSON. In comparison to XML, JSON has a small footprint and a more flexible structure. The latter can, but not must, be a disadvantage.

HTML5 is the latest version of HTML, which defines new elements and additional JavaScript APIs. Some of these APIs can be an advantage for MVC-based web applications, including:

- The *Web Workers API* relaxes the single-threaded client-side JavaScript model to support multiple threads, called "workers". Workers are to classic operation system threads; they do not share memory (thus, they lack access to the DOM) and can only communicate with the main thread through asynchronous messaging (Flanagan 2006, pp. 680–687). Nevertheless, Workers make it possible to execute multiple tasks in parallel without blocking each other or the user.
- Using the *WebSocket API*, long-living TCP connections can be established. When using AJAX to communicate over the network, HTTP is always involved, which is a stateless (session–less) protocol. Opposed to that, WebSocket connections can be persistent and allow data to be sent and received as long as the socket is open. As WebSockets bring their own protocol on top of the Transmission Control Protocol (TCP), the server being queried also needs to support that protocol. There exist

several libraries to add support to web servers, for example an Apache and a Node.js module (Flanagan 2006, pp. 712–716).

When used together with Web Workers, WebSockets can provide a web application with threaded network functionality. This is also relevant for MVC applications with real-time Model synchronization, as WebSockets can be used to notify the client of an updated server-side Model. This is further discussed in Section 4.1.2.

- The client-side *Storage APIs* allow an application to store structured data within the web browser's sandbox. Before, only HTTP Cookies could be employed for client-side data storage, but Cookies are limited in size and they allow only textual data to be included. One of the new APIs is *IndexedDB*, an object database that is currently supported in Firefox and Chrome, whereas the relational *Web SQL* database is implemented in Chrome, Safari and Opera. The *Filesystem API* allows web applications to create, delete and manipulate files and directory structures in a sandboxed environment. All web applications with the same origin (host, port, and protocol) share one filesystem — this is a concept borrowed from HTTP cookies to avoid unauthorized access to data from the wrong web application. (Flanagan 2006, pp. 700–712)

  These APIs can be used to allow advanced storage of data, breaking with the 4KiB limitation of HTTP cookies. They can be used for caching and client-side persistent storage. It is even possible to write offline applications using these APIs.

3.2 Client–Server Architecture on the Web

This section describes the general architecture of 2–tier web applications, and then discusses the terms "Thin Client" and "Rich Client" in this context.

The first tier, also called "client", is the end user's web browser. The second tier, the "server", is a web server on the internet or inside a local network, for example a company's intranet. The key technology for this architecture is the HTTP protocol that has to be understood by both client and server. Thus, the web browser acts as an HTTP client and the web server acts as an HTTP server.

The common interaction between the user, a client and server using HTTP is further described by the following sequence diagram in Figure 3.1

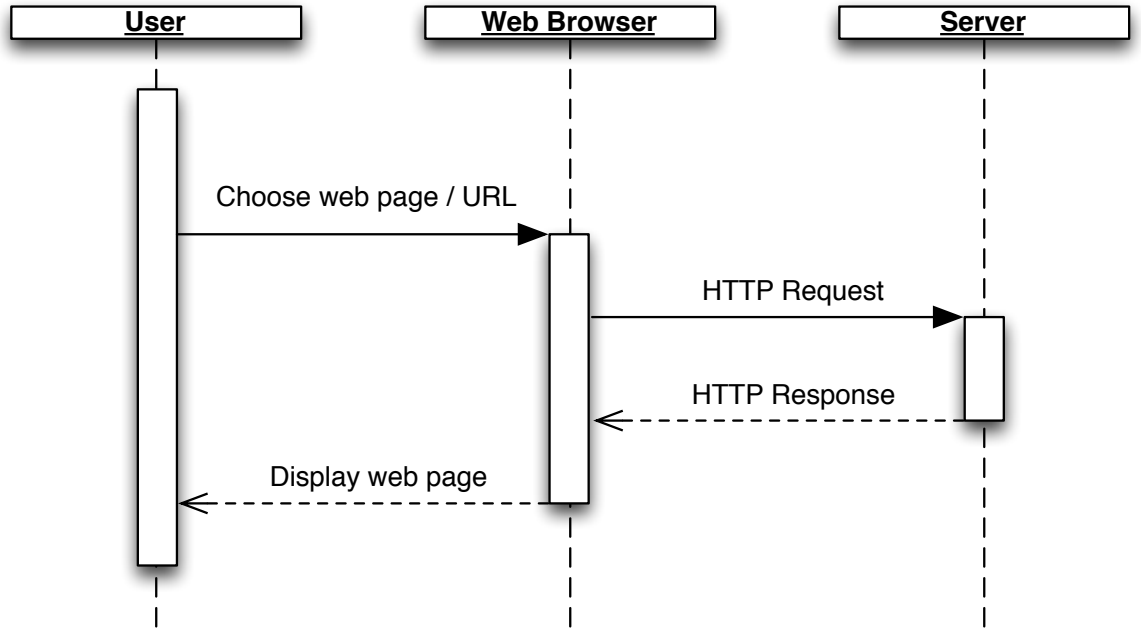

Figure 3.1: Sequence diagram of a User–Client–Server interaction

The actual HTTP request is illustrated in Listing 3.1. Such requests have to be made for every resource loaded from the server, such as HTML, JavaScript, images and stylesheets (CSS). As every request includes protocol overhead, the number of requests to be made is an important factor to the performance of a web application. This is further discussed in Section 4.1.3 and Section 4.2.

```
GET /index.html HTTP/1.1
Host: www.example.com

HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Content-Length: 438
Content-Type: text/html; charset=UTF-8

... Response body ...
```

Source: Wikipedia (n.d.b), modified by the author

Listing 3.1: HTTP conversation

3.2.1 Terminology

In computer technology, the terms "Thin Client" and "Rich Client" (also: "Thick Client" or "Fat Client") have multiple meanings, so it is necessary to define their meanings in this context.

These two terms often describe computer hardware. A Thin Client in the hardware context is a computer that heavily depends on another computer (e.g. a server) to fulfill its computational role, whereas a Rich Client not only handles input and output (I/O), but also processes data.

This concept of dependance also applies to web applications. In that context, the Thin and Rich Client are not computer hardware, but rather represent the client side of a web application (thus, they are software running *inside a browser*).

The term "Rich Internet Application" also is sometimes used for web applications with a Rich Client, but it more often refers to applications powered by browser plug-ins, such as Adobe Flash or Adobe Flex.

The following two definitions are kept generic and reflect the fact that there are different variations in the range from a Thin to a Rich Client. Four of those variations are further discussed in Section 4.2 with respect to an MVC architecture. Both definitions are examined from both a user and technology perspective.

3.2.2 Thin Client

We can also refer to the Thin Client as the classical way of writing web applications. From a user perspective, a Thin Client feels like a usual web site. Every time the user switches the perspective or navigates to a different part of the web site, the URL changes and a whole page is loaded from the server. This is, for example, the fact with http://www.craigslist.com/[1]. If you switch between categories at the Craigslist web site, the whole page gets loaded and rendered again.

This is due to the fact that in many Thin Clients, every page of a web site either has its own HTML file, or gets preprocessed on the server (using PHP, ASP, JSP or any other preprocessor) depending on defined parameters[2]. From a technology perspective, a page is — when requested — constructed on the server side and sent over to the client side. There are no subsequent data

[1] Craigslist is a classifieds platform extremely popular in the United States.

[2] These parameters may, for instance, be GET or POST parameters of the respective HTTP request (Fielding et al. 1991).

loaded (except for resources already specified in the original page). The page is dynamically generated, but the web application is not interactive.

From the perspective of software architecture, a Thin Client is static and "dumb". Most of the business logic and data — and thus, most of the MVC structure if this is the used architectural pattern — is on the server. The content gets prepared on the server, processed with real data and is then sent to the client. However, the client does not know where the data are from and how to retrieve additional or updated data. Also, it is not able to further process the data without communicating with the server. This means that a user can modify data on the client side, but these data must be sent back to the server side to be processed and then the updated page has to be retrieved again.

The sequence diagram (Figure 3.2) shows the process of a user requesting a web page and then updating data using a Thin Client / Rich Server with an additional database tier. Business logic, for example input verification, is executed on the server. After updating the data, the whole web page has to be preprocessed on the server, sent to the client and displayed again. The browser and user are inactive during this time.

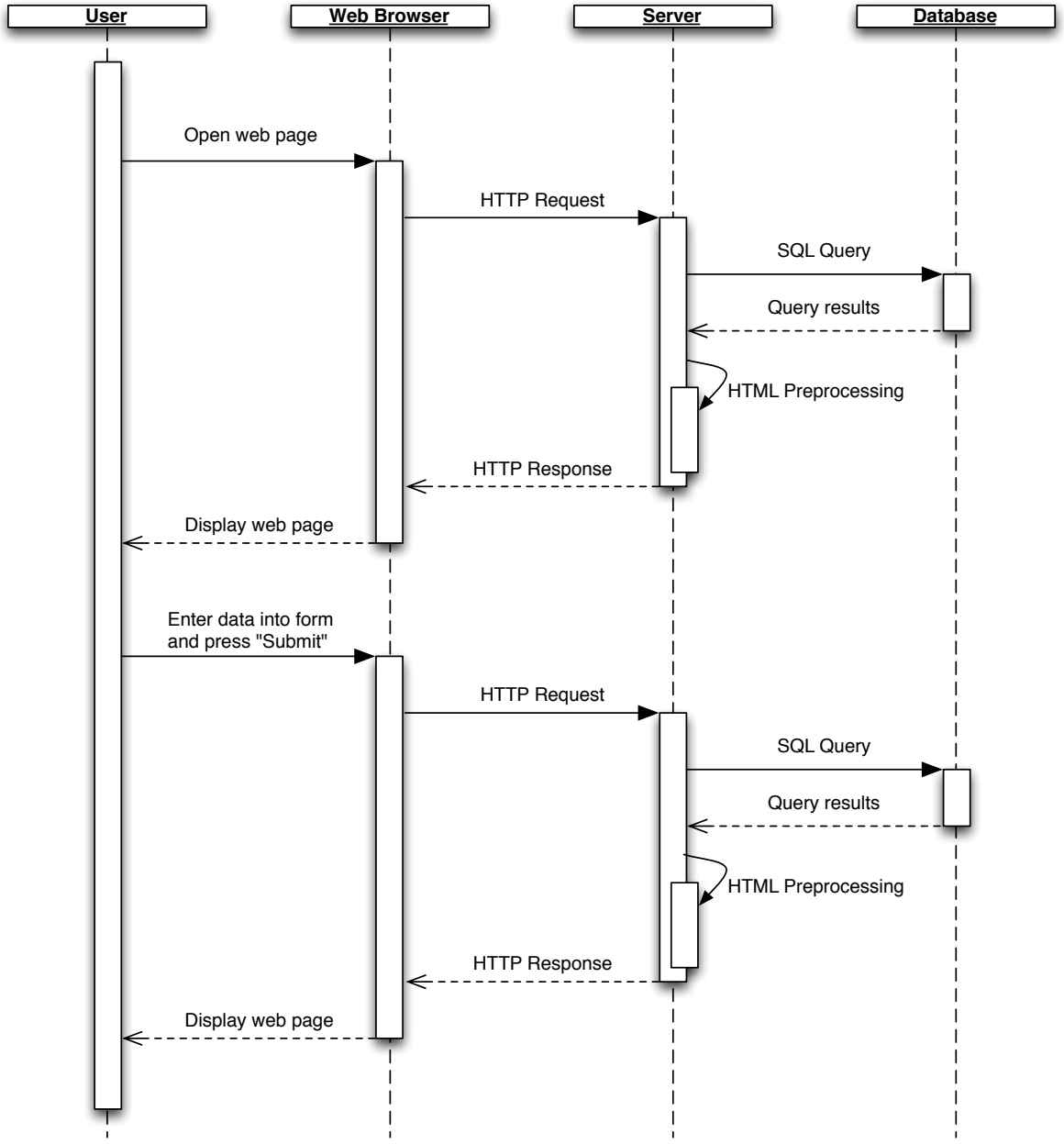

Figure 3.2: Sequence diagram of a Thin Client user interaction

An illustrating example for a Thin Client / Rich Server model are the two variants of Model–View–Controller in JSP–based web applications, called *Model 1* and *Model 2*. As described by Johnson (2003, pp. 444–446), Model 1 provides individual JSP pages for every section of the web application, mapping those pages to EJBs. The flow of navigation is directed by links to the different JSPs, which act as both Controller and View. The JavaBeans are the respective Models.

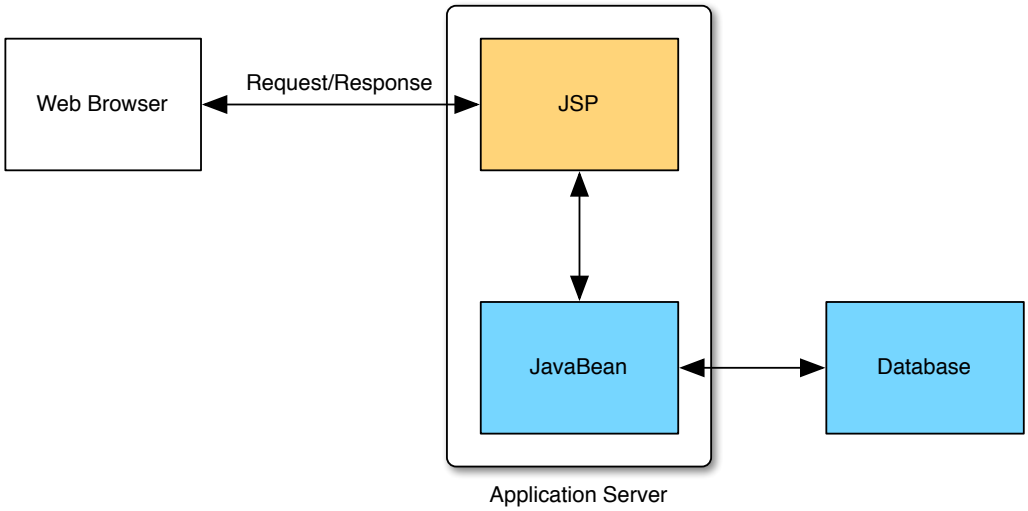

Figure 3.3: JSP Model 1 Architecture

In Model 2, the Controller part is encorporated by one central Servlet acting as an entry point into the application (Johnson 2003, pp. 446 f.). It handles all browser requests and manages the application state. Depending on the request and the state, it selects a View (JSP) to serve back to the browser. Similar to Model 1, the View communicates with the Model (JavaBean), but the JavaBean is modified by the Servlet instead of the JSP. Having only one Controller differs from the Smalltalk-80 definition of MVC, but can be seen in some interpretations as an *application controller*.

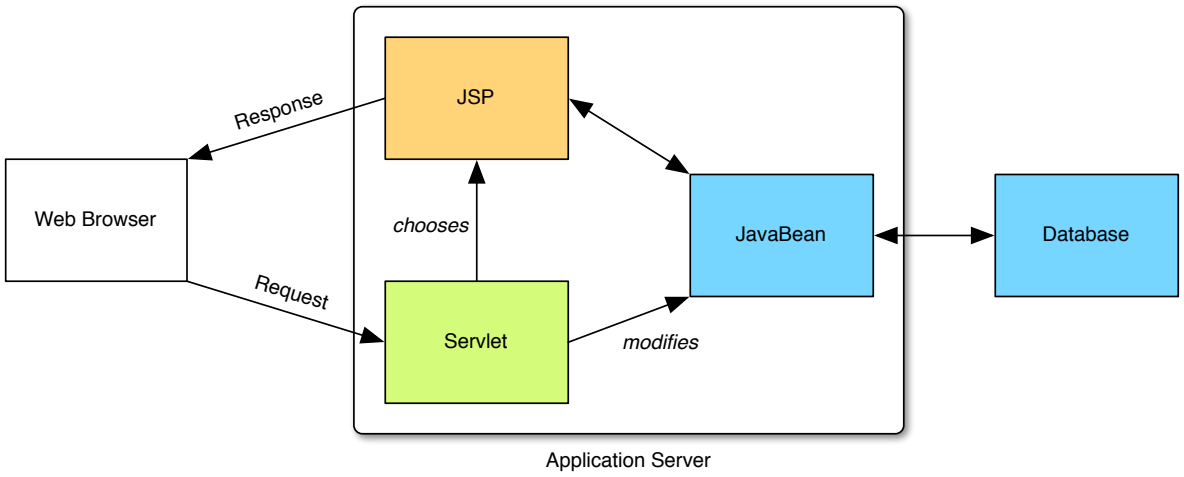

Figure 3.4: JSP Model 2 Architecture

3.2.3 Rich Client

From the user perspective, the key difference to a Thin Client is that the Rich Client acts *like a desktop application rather than a website*.

From a business logic perspective, a Rich Client web application runs as much application logic as possible on the client side. This means, data transformation (like sorting, filtering, conversion) and referencing happens on the client side, as well as changes to the data (which then are made persistent on the server side). There still are parts that have to be executed on the server, for example security (authentication, authorisation) and the connection to a third tier (such as a database).

From a technology perspective, a Rich Client makes extensive use of JavaScript and the AJAX technique. This can, for example, happen through the use of a client-side MVC pattern that has not only the View, but also the Model and the Controller running in the browser. In modern scenarios, another example is the use of Web Sockets that lets the server-side Model notify the web browser when changes happen. The key point is to reduce communication with the server to a minimum.

Figure 3.5 shows a sequence diagram of a user interaction with a Rich Client / Thin Server. The same activities as in Figure 3.2 are executed, but due to the asynchronousness of AJAX calls, the browser and user can still interact during the second request. As indicated, the web page gets only preprocessed once (when first requested). After the form submit, the processing of the returned, updated data happens in the browser, and not on the server as in Figure 3.2.

Also, business logic such as input verification is executed on the client-side already (before the form is submitted). However, to ensure data consistency and security, in most implementations the submitted data — as well as the user's authorization — is verified on the server again.

In extremely client-sided web applications, such as IBM Content Navigator, there happens no preprocessing on the server at all. On first request, the server only sends one single, nearly empty HTML skeleton file. The client loads all needed functionality (mostly Dojo and IBM Dojo Extensions (IDX)) via AJAX, building the GUI solely using JavaScript. There are no more "pages" in the sense of web pages; navigation is done using JavaScript. If the user navigates to a different application pane, Content Navigator swaps out the respective DOM nodes, which gives the feel of a real desktop application. Content Navigator is further discussed in Chapter 5.

This navigational pattern is described by Osmani (2012, pp. 104 f.) with one main characteristic: the state of the application is managed on the client side, using a component that has the role of a "router". In Thin Client applications, the state is managed on the server side using sessions[1].

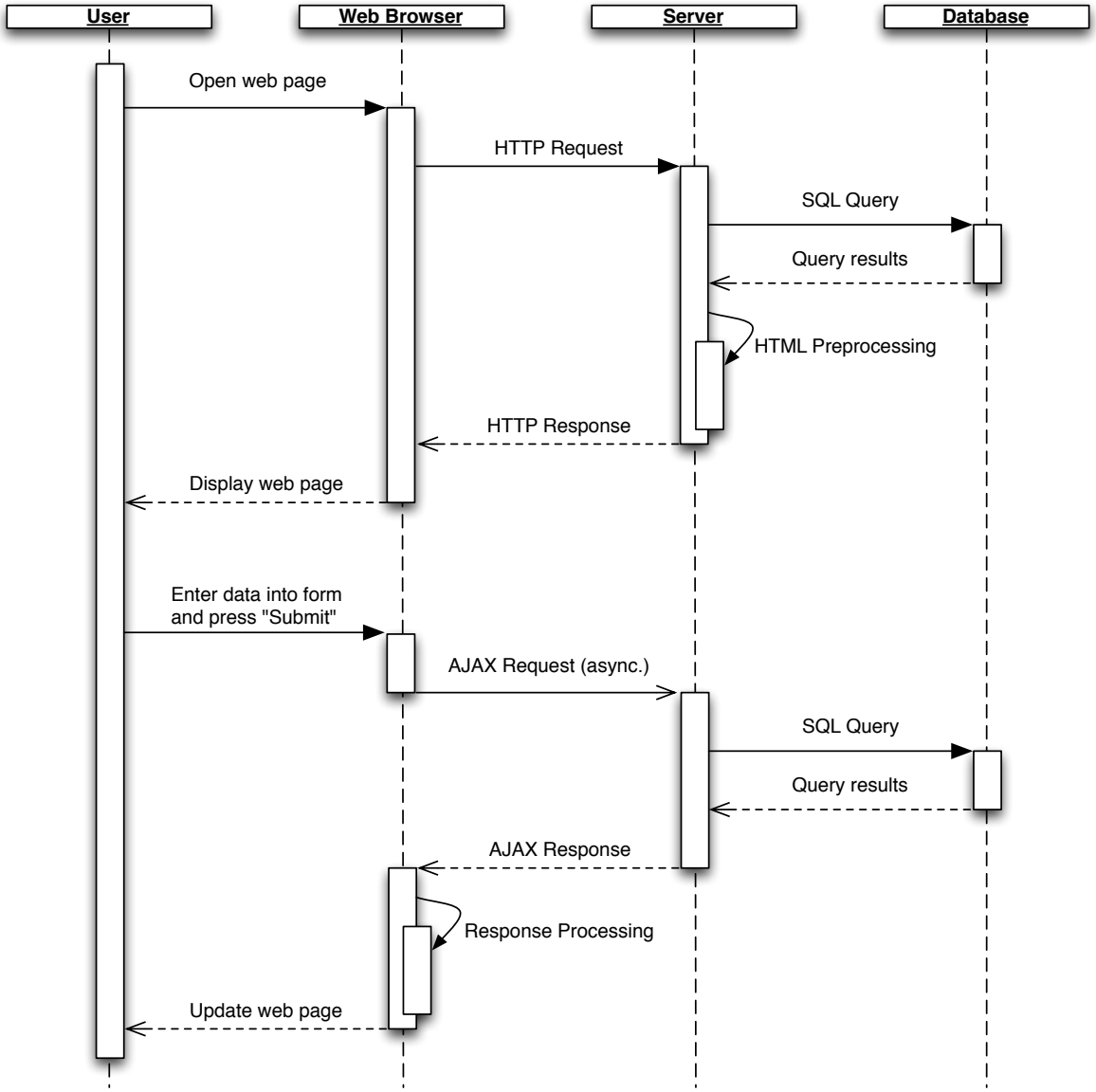

Figure 3.5: Sequence diagram of a Rich Client user interaction

[1] A identifies interrelated requests using a session ID (usually a unique combination of alphanumeric characters) to provide state on top of the stateless HTTP protocol.

4 Applicability of Different MVC Architectures for Web Applications

In the previous two chapters, the Model–View–Controller pattern as well as technologies and techniques relevant to web application architectures have been presented. The challenges that result from applying MVC to web applications are discussed in this chapter by means of three defined criteria and four different MVC architecture variations.

4.1 Evaluation Criteria

In this section, three criteria — *data model complexity*, *real-time model synchronization*, and *data volume* — are defined to help evaluating the applicability of different possibilities to implement MVC in web applications. These criteria are accompanied by problems that have to be solved by an appropriate architectural design.

Some of the criteria are quantitative and can thus be evaluated objectively, such as *Data Volume*. Others, such as *Data Model Complexity*, are rather intangible and thus have a high interpretability. Therefore, examples are given to illustrate each criterion.

4.1.1 Data Model Complexity

Data model complexity is a criterion that cannot be measured objectively, and is also hard to define. It is an inherent problem for many domains that needs to be solved by an adequate web application architecture.

The complexity of a data model can best be illustrated by means of a relational database. The more the data model is normalized, the more complex it is. Normalization of data means to minimize redundancy and dependency. Normalized data often have cross references — foreign

keys in relational databases, object references in object–oriented programming languages and databases.

For Model–View–Controller based web applications, complex data models lead to a large number of Models and references between different Models. This can have challenges for the software design on the one hand, and for maintaining data consistency on the other hand.

Examples

A simple *task application* with the only domain data being "tasks" has no high data model complexity. There usually is an additional Model for the users, but this never has to be managed on the client side. Users do not interfere with each other in this application, and they can only access their own tasks. A good example for an application of low data model complexity is *Do It (Tomorrow)*[1].

A project management application that manages users and roles, teams, projects, tasks and milestones and provides both an end-user as well as an administrator interface, has a high data model complexity.

IBM Content Navigator also is an illustrative example for web application with a complex data model. There are not only nested and interconnected Models, but also different data sources for the Models. Representational State Transfer (REST) services can either be directly connected using Dojo, or they are accessed using the service layer of ICN as a proxy. In the latter case, Dojo interfaces with a service that acts as a REST API and redirects every request to the actual REST server after applying advanced business logic — such as authentication and authorization. This can, for example, be used for Single Sign-On (SSO). Other data sources, such as the content repositories FileNet P8 and ContentManager, also have to be included. The fact that Content Navigator supports multiple repositories through SSO in one single interface leads to an even higher complexity. ICN is described in Chapter 5 in more detail.

4.1.2 Real-time Model Synchronization

In the area of web applications, the term *real–time* is not comparable with "real–time computing", and should not be mistaken for it. A web application that supports real–time Model synchronization makes sure that changes to a server-side Model, which are taken by one client

[1] See http://tomorrow.do/

instance, are immediately pushed to all other client instances, too. Neither are those clients required to reload the page, nor do they have to request the updates themselves.

In opposition to this situation, the classic *pull* mechanism is usually built as a timer. A timer is a (JavaScript) function that gets executed periodically, for example "every 5 minutes", and pulls updates from the server to the client. For real-time Model synchronization, *pull* can no longer be the technique of choice, as there might happen changes to a Model in between two executions of the timer. Instead, there are two possiblities for the server to notify the client of changes:

- is an umbrella term for a set of techniques that allow low-latency data transfers in web applications. Two of them are commonly used: *long polling* and *forever frame*.

  Long is "a technique that optimizes traditional polling to reduce latency." (Schiemann 2007*b*) Whereas traditional polling means to send requests in defined time frames (for example "every 15 seconds") and to get an immediate response, long polling is not responded to until there actually *is* something to respond with (e.g. updated data). This kind of request is also called long-running request. As soon as the HTTP respond returns, it is processed on the client, and another long-running request is sent immediately. The same happens if there is a timeout. Long polling makes use of the `XmlHttpRequest` JavaScript object.

  The "forever frame" is a different approach: an iframe[1] is used to incrementally receive data, based on an HTTP 1.1 feature called *chunked encoding*, as described by Schiemann (2007*a*). Using one single, long-lived HTTP request, data can be received whenever necessary, as they are sent in chunks. The connection is not closed until the server decides to close it.

  Both of these approaches — and in general every Comet technique — work asynchronously. This is essential for web applications that require real-time data synchronization, as user interaction would otherwise be blocked. On the server this problem can be solved using non-blocking I/O or threads.

  There are Comet implementations for most JavaScript frameworks, such as *cometD* for Dojo[2] and the comet plug-in for jQuery[3].

- The new HTML5 APIs provide the possibility to create long-lasting TCP sockets in the browser, called WebSockets. Using a WebSocket, a permanent, duplex connection can

[1] An (*inline frame*) is an HTML element that makes it possible to embed an HTML document into another one.
[2] See http://cometd.org/
[3] See http://archive.plugins.jquery.com/project/Comet

be established to the server. Updates that happen on the client can be sent using this socket, and changes that are made on the server (or pushed to the server using other client instances) can be received.

This concept is especially useful if combined with Web Workers, so that the socket is not blocking any user interaction or other application functionality.

The great advantage over the *forever frame* technique is that WebSockets work in both directions: not only can the server send information to the client, the client can also send additional information to the server (for example to abort the running request and send a different one).

Examples

Web-based communication tools, for example instant messaging programs, serve as good examples for the real-time criterion. The chat tool integrated in many of Google's products, *Google Chat*, uses push techniques. Which techniques are used exactly is not publicly documented, but with respect to Google's leading role[1] in web technology development, it can be assumed that Google Chat uses WebSockets if available, and Comet techniques otherwise.

Other typical applications for a real-time synchronization scenario are collaboration tools. Google Docs[2], for example, is an online office suite that provides spreadsheets, documents, presentations and more. These documents can be edited by multiple users in real-time, which means you can see the changes another user makes right as he types, and vice-versa.

4.1.3 Data Volume

This criterion describes the volume of data a web application has to process. Data volume is critical to application speed and responsiveness.

It is assumed that the data volume described here has to be processed *on the client*. In other words, the user has to get in contact with these data in some way, either by creating, manipulating or viewing them.

[1] Amongst others, Google develops the open source JavaScript engine V8, contributes to the WebKit HTML rendering engine and is part of the W3C (the comittee that develops web standards such as CSS and HTML).

[2] Now part of Google Drive, see https://drive.google.com/

High data volumes are causing two significant problems that can slow down a web application:

- Long network transfer times force the user to wait for data before he can work with the application.
- A large amount of data in the browser memory can cause bad performance in older browsers or on older machines.

Both problems lead to bad usability of the web application. To solve these problems, network traffic between the application's client-side and server-side has to be reduced to a minimum. There are several techniques to achieve this, including the following two:

Lazy loading is one approach to reduce both network traffic and the amount of data being held in browser memory. It is based on the assumption that not all the data have to be present on the client at once. Lazy Loading means loading data "on demand". Usually, it is implemented using some sort of prediction strategy which predicts what data will have to be loaded next. This way, the user does not need to actively request additional data.

Lazy Loading needs a client-side Model, or at least an instance that manages the state of data that are already loaded, so it can be determined which parts of the data still need to be loaded.

Although the actual Model may be quite large on the server, lazy loading keeps it as small as possible on the client and leads to a more performant application.

Caching can be leveraged on different network nodes, for example using a proxy, web server or browser cache. But also web applications can implement caching. Using the HTML5 *localStorage* API, data can be made persistent on the client side using an associative data model[1]. Other storage APIs, such as IndexedDB or WebSQL can be used too, but are not as widely supported by browsers as localStorage.

In addition to the techniques used for the data volume of Models, also the data volume of source code can be reduced. This can be achieved by packaging code into logical, independent *modules*. In opposition to server-side programming languages like Java, , Python or C++, JavaScript — as of version ECMA-262 — does not provide a way to import and use modules.

[1] Associative arrays map a certain value to a key, and are therefore also called "key–value stores" or "dictionaries".

A solution to this is Asynchronous Module Definition (AMD)[1], an API for packaging code into modules and loading them asynchronously, on demand. It is supported by RequireJS, Dojo as of version 1.7 and other JavaScript frameworks and libraries.

The amount of data to transfer from the server to the client is usually smaller if using AMD, as only the parts of the application are loaded that are really needed. This leads to shorter loading times. Additional components, which were not needed at the initial loading of the application, can be loaded as their functionality is requested (known as “”).

Examples

Twitter[2] is an application with very high data throughput: 200 million Tweets were sent per day as of November 1st, 2011 (over 2300 per second)[3]. Although every user can only see a small fraction of these, the Twitter stream cannot show *all* Follower's Tweets at once. Therefore, the Twitter web application as well its Android and iOS applications all support lazy loading for the list of Tweets. It only loads a limited number of Tweets on initial page load, but loads additional (older) ones when the user scrolls down the page. Twitter shows a loading animation (see Figure 4.1), although this animation should not be seen by users, due to predictive loading (Tweets are loaded *before* the user reaches the bottom of the page). To make this screenshot possible, the internet connection was shut down temporarily.

Figure 4.1: Lazy loading on Twitter, indicated by a loading animation

[1] See https://github.com/amdjs/amdjs-api/wiki/AMD

[2] Twitter is a web application to exchange short messages, so-called *Tweets*, with other people (*Followers*) who have subscribed to you.

[3] See https://dev.twitter.com/discussions/3914

A second illustrating example for handling of high data volumes is *Backbone.js*. This is a JavaScript framework for structuring client-side web applications according to an adaptation of the Model–View–Presenter pattern. It uses RESTful calls via AJAX to synchronize its client-side Model with the server. These calls can be replaced by a localStorage adapter[1], so that localStorage is employed as the persistence layer.

Models in Backbone.js are single objects that keep a record of domain data. To store more than one record, a Collection is used[2]. Backbone.js allows to assign the persistence storage on a per-Collection basis.

Using two different Collections of the same Model allows the developer to implement a cache mechanism for Model data. One Collection is used to connect to the actual persistence layer, for example a REST server, whereas the other one connects to the localStorage. If Model data are requested, the application can first look for them in the localStorage Collection; if they are not present there, they are requested from the REST Collection and copied to the localStorage Collection for caching. As in every cache mechanism, the implementation has to make sure that the cached data are up–to–date.

Listing 4.1 shows the definition of a "Person" Backbone.js Model including default values and input validation. The "People" Collection of "Person" Models is tied to the localStorage API for data persistence. To illustrate the usage of Models and Collections in Backbone.js, a "Person" is instantiated and saved into the Collection.

[1] See https://github.com/jeromegn/Backbone.localStorage

[2] This is a matter of terminology. "Model" often refers to a whole set of records, but the Backbone.js developers decided to use this term for a single instance and the term "Collection" for multiple instances.

```
var Person = Backbone.Model.extend({
  defaults: {
    salutation: "Mr.",
    age: 18,
    children: []
  },
  validate: function(attributes){
    // If validate() returns a string, Backbone throws an error
    if(attributes.age < 0) {
      return "You can't be negative years old";
    }
    if(name === undefined)
      return "You need to have a name";
    }
    // This requires the jQuery library
    if(!$.inArray(attributes.salutation, ["Mr.", "Mrs.", "Ms."])) {
      return "You need to have a proper salutation";
    }
  }
});

var People = Backbone.Collection.extend({
  // Save in the localStorage repository "People"
  localStorage: new Backbone.LocalStorage("People"),
  // Use the Model defined above
  model: Person
});

var bruce = new Person({
^^name: "Bruce Wayne",
^^age: 32
});

var age = bruce.get("age"); // 32
var name = bruce.get("name"); // "Bruce Wayne"
var salutation = bruce.get("salutation"); // "Mr."
var children = bruce.get("children"); // []

var localCollection = new People();
localCollection.push(bruce);
```

Listing 4.1: Backbone.js Model and Collection using localStorage for persistence

4.2 MVC Architectures in Web Applications

The Thin Client and Rich Client architectures introduced in Section 3.2 are general descriptions of frequently implemented web application architectures, independently of Model–View–Controller or any other architectural pattern. If in the software engineering process a pattern of the MV* family is chosen for a web application architecture, some decisions are yet to be made. There are several ways to implement Model–View–Controller on client as well as on server side, as the components can be distributed on both network nodes differently. This section discusses various solutions, along with synchronisation strategies and possible implementations.

Controller and Router

It is necessary to clarify the terms “Controller” and “Router” in this section to avoid misunderstandings. The design pattern *Front Controller* or (both terms describe the same pattern) were already introduced on page 13. According to Osmani (2012, p. 105), routers are “neither a part of MVC nor present in every MVC-like framework”. This is definitely true for Model–View–Controller as a pattern, but when implementing it in a 2–tier web architecture, the router is an inevitable part, as it is a Controller “that handles all requests for a Web site” (Fowler 2002, p. 344).

The Router has a responsibility that is originally assigned to a regular MVC Controller: it reacts on user input. On thin clients, the only user input recognizable by the application is coupled to HTTP requests. This includes navigation on the one hand (the user clicking on hyperlinks) and submitting form data on the other hand. Both actions are tied to URLs, so the processing of the according requests — which are in the responsibility of a Controller — is done by a Router.

An excellent example for this is Model 2, a Java EE implementation of MVC, which uses a Servlet as the Front Controller (see page 32). This Servlet chooses actions to take and the View (JSP) to display based on the URL, which is both a Router’s and Controller’s task.

As a conclusion, it can be assumed that routing in web applications is part of the server–side Controller, which is the reason why the diagrams in the following section do not show a distinct Router component. The first three MVC web architectures handle routing on the server, as they all include a server–side Controller.

On Applicability

The applicability of the presented MVC architectures is assessed on the basis of the previously defined criteria. For each of the four architecture variations, a table is given to evaluate the applicability of an architecture with respect to one of the three criteria. For each criterion, the result can be

- *Yes*, which means that this architecture is suitable for web applications with the respective requirement, as most arguments are in favor.
- *No*, which means that this architecture is not suitable for web applications with the respective criterion, according to most of the presented arguments.
- *Depends*, which means that there are balanced arguments speaking for and against the applicability of this architecture.
- There are always exceptions to these evaluation results.

The choice of the right architecture for an application depends on many factors. The applicability tables created here are not absolute, but provide an objective (as in "supported by arguments") basis to compare the four architecture variations.

4.2.1 Server Side MVC Architecture

The classical and most simple solution is to implement the MVC stack completely on the server side. This would be the case in a Thin Client as described in Section 3.2.2.

The View in this variation is usually an HTML file, created using an HTML preprocessor or a templating language[1]. It gets preprocessed on the server, is then sent to the browser and not modified after that. The user interacts with the MVC application through the web browser, which is why there is no real client–side View, but only the DOM.

When using an object-oriented programming language on the server, the Models can be implemented interfacing an object-relational wrapper or the API to an object-oriented database like MongoDB or CouchDB. A Model then would basically be a class that utilizes Create, Read, Update, Delete (CRUD) methods. In procedural programming languages, the Model usually is

[1] A templating engine is a software that processes a text file, replacing placeholders with actual values that are valid for a given situation. In addition to placeholders, constructs of programming languages — like conditionals and loops — can often be used too, hence the term *templating language*.

the database itself. It can be manipulated and queried directly from the programming language, for example using the Structured Query Language (SQL).

The Controller is the application code inside the View, or included in the View through external files. It is code that decides what actions to take and with which values to fill the template's placeholders, depending on application state and user input.

Following the description above, the components in a *Server Side MVC Architecture* can be very closely coupled, which contradicts the idea of *Separation of Concerns*. Frameworks, such as the two described below, try to enforce decoupling.

Structure

Figure 4.2 shows the architecture of a full server-side MVC web application.

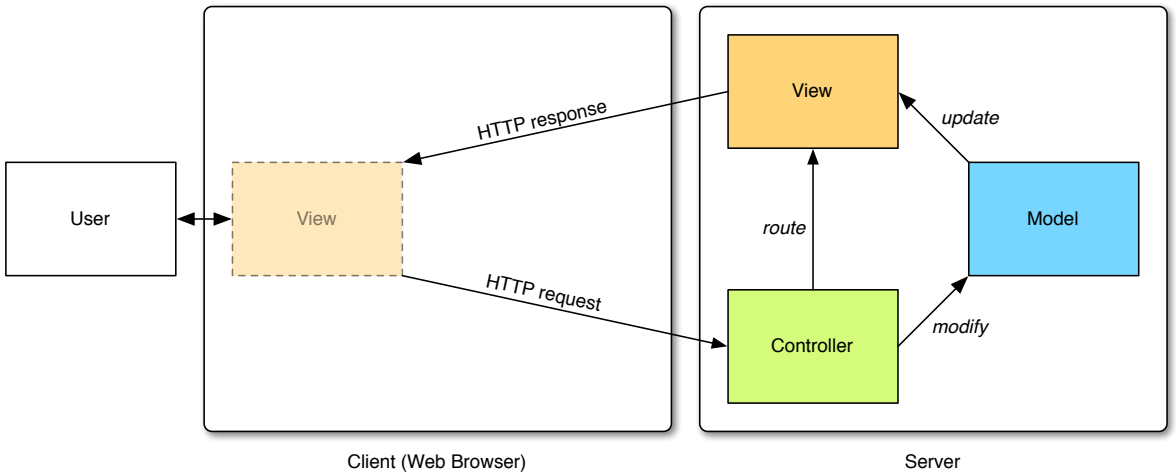

Figure 4.2: Structure of a Server Side MVC Architecture

The diagram reflects the sequence of control in this MVC scenario. It is assumed that the server-side View is bound to a URL, and thus is the HTML document to receive. Loading of a URL is always initiated by the user through the web browser.

The client-side View is displayed slightly transparent, as it is not a real MVC View that gets data from a Model and performs actions through a Controller. It is the displayed version of the web page that was loaded in the browser, and it acts as a mediator between the user and the server (following URLs after the user clicks on links and displaying a web page after it is received). In

later presented architectures, the Controller is mapped to a URL and invoked by the client-side Controller, instead of by the server-side View.

Implementations

Common implementations for this variant include MVC frameworks for server-side programming languages:

Struts is an open source framework for building web applications using Java EE. It implements a *Model 2* architecture, as described in Section 3.2.2. Struts uses a Servlet as the Front Controller to route requests to the actual Controller, which is responsible for a certain request. It responds with JSPs as Views and uses JavaBeans as Models.[1]

Ruby on Rails — or just *Rails* — is a MVC web application framework for the Ruby programming language. It can make use of a variety of templating systems (eRuby, HAML and others) to construct Views. For the Models, Rails uses an object-relational wrapper on top of a relational database (such as MySQL, SQLite, DB2, and Oracle), but extended through plug-ins it can also use object databases for persistent storage of domain data.

ships the JavaScript framework *Prototype*, which makes it possible to build AJAX applications using Rails as the backend, but Rails itself is a pure server-side MVC framework.[2]

Applicability

Criterion	Applicability
Data Model Complexity	*Yes*
Real-time Model Synchronization	*No*
Data Volume	*No*

Table 4.1: Applicability of a Server Side MVC Architecture

The *Server Side MVC Architecture* can be used for applications with a complex data model. As the pattern components reside completely on the server side, maintenance is easier to perform than if it was distributed. MVC frameworks such as Struts, Spring, FLOW3, Rails and Synfony

[1] See http://struts.apache.org/
[2] See http://rubyonrails.org/

offer data modelling possibilities that are powerful enough to reproduce even very complex data models.

It is not suitable for real-time Model synchronization. Once sent to the client, the page cannot be updated — it remains static. Providing real-time updates, or even non-real-time updates, would require a client-side mechanism to request and process subsequently sent data. Such mechanism is not present in a Server Side MVC architecture.

High data volume turns out to be a problem for this architecture, too. As data cannot be fetched after the site is loaded, all data that the user could want to process have to be loaded initially, and again with every navigational action the user might take. This results in high network traffic.

4.2.2 Distributed Controller MVC Architecture

The first change that can be made to the full server-side MVC is to push the Controller to the client-side. This does not mean that there is no Controller on the server anymore, but its responsibility is reduced in comparison to a full server-side one. Thus, this MVC web application architecture can be called *Distributed Controller MVC Architecture*.

In Server Side MVC Architecture applications, navigational actions are bound to different pages. Refreshing a View requires to refresh the web page, and manipulating the Model is usually bound to calling a URL with POST or GET parameters — which is the Controller's task.

In Distributed Controller MVC applications, the navigation is not only bound to different *pages*. Still, pages can be accessed through the URL and lead to different main areas of the application. This task of routing is done by the server-side Controller[1]. The client-side part of the Controller, on the other hand, is encorporated by the possibility to interact with the web application *without* changing the page (and thus loading a new URL) each time. It is realized as JavaScript code running in the browser, which is capable of handling minor navigational patterns within the page and delegating Model manipulation to the server–side Controller using AJAX. These tasks can include lazy loading of content when scrolling or the creation of popup windows to display more detailed data.

Manipulations on the Model are initiated from the client-side Controller using an AJAX call, but are carried out by the server-side Controller code. This is usually a Servlet or a script, as described for Server Side MVC on page 44. The server–side Controller needs to be able to analyze the AJAX request and act accordingly. The *Distributed Controller MVC Architecture* does not maintain a client-side Model.

Structure

The following architectural diagram (Figure 4.3) illustrates the distributed Controller component.

[1] See p. 43 for more information on the connection between *Controller* and *Router* in web-based MVC applications.

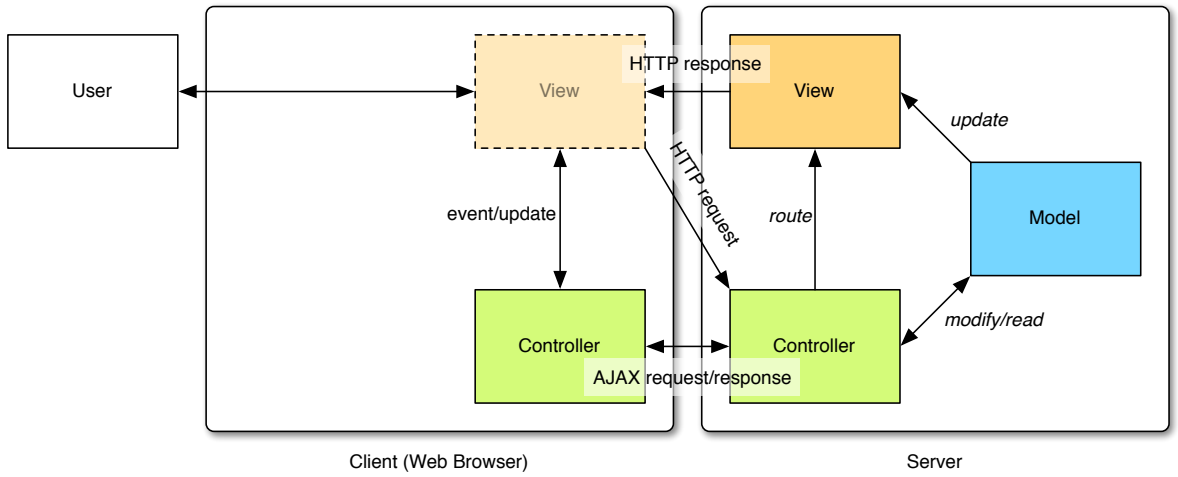

Figure 4.3: Structure of a Distributed Controller MVC Architecture

Implementations

The web application implements a Distributed Controller MVC Architecture, which is described by Selvitelle (2010).

The application's main areas ("Home", "Connect" and "Discover") as well as users' profiles have their own URLs. Thus, the application has a server-side Controller. However, there is a lot of functionality that can be used without switching pages.

For example, when looking at a Twitter stream (or a search), you can scroll down to the bottom of the page, and additional (older) are loaded. The respective event, in natural language "the user scrolls to near the bottom of the page", is processed by a client-side Controller, which then sends an AJAX request to the server and inserts the results (more Tweets) into the View (the list of Tweets, or *stream*). This can be examined by logging the HTTP activity of Twitter.

According to Venners (2009), the Twitter web application is based on Ruby on Rails as the server-side MVC framework. For the client-side, there is no MV* framework used. Twitter only includes the jQuery JavaScript library, which is used for event handling and AJAX requests (client-side Controllers). This is proven by the JavaScript source code used by Twitter.

jQuery Example

To illustrate the interaction of client-side and server-side Controller, the following listings show how data from a form can be made persistent on the server without reloading the page. A simple task application using the JavaScript library *jQuery* serves as an example.

```
<form>
  <input type="text" />
  <button type="submit">Add new Task</button>
</form>
<ul>
  <li>Wash the dishes</li>
  <li>Do laundry</li>
  <li>Grocery shopping</li>
</ul>
```

Listing 4.2: Task list HTML skeleton (View)

The HTML skeleton in Listing 4.2 contains a form with an input text field and a submit button, respectively. Also, it contains a list of items that represent already existing *tasks*.

```
$('form button').click(function (event) {
  event.preventDefault();
  $.ajax({
    url: "newtask.php",
    data: {
      task: $('form input').val()
    }
  }).done(function() {
    $('<li>').html( $('form input').val() ).prependTo('ul');
    $('form input').val('').focus();
  });
});
```

Listing 4.3: JavaScript-based client-side Controller for the task list

The JavaScript code in Listing 4.3 represents the client-side part of the Controller. An anonymous function is registered as a callback for the *click* event of the form's submit button. In the callback itself, the default behaviour (that would be to submit the form) is prevented using

`event.preventDefault();`. Then, an AJAX request is sent to the URL `newtask.php`, providing the text inside the form's input field as a parameter. By default, this is an asynchronous call using the HTTP GET method.

When the call returns successfully, another callback (the anonymous function inside the `$.ajax().done()` invocation) adds the new task as a list item on top of the existing list. It then clears the input field and gives the keyboard focus back to it.

The PHP code of the server-side Controller is not listed here. It would typically insert the given task into a database and return a unique ID, which then could be processed on the client.

Applicability

Criterion	Applicability
Data Model Complexity	*No*
Real-time Model Synchronization	*Yes*
Data Volume	*Depends*

Table 4.2: Applicability of a Distributed Controller MVC Architecture

The Distributed Controller MVC Architecture makes more use of client-side programming and thus is applicable for web applications that require more dynamicity on the client-side than the Server Side MVC Architecture.

The fact that this MVC distribution allows additional data loading without maintaining a client-side Model makes it a good choice for simple data structures. Complex Models, on the other hand, are harder to implement using the Distributed Controller MVC, as the missing client-side Model makes it impossible to manage data in a structured way. It is sufficient for the Twitter web application, as its data model is rather simple.

The client-side Controller allows loading data at any time, so it is possible to build applications with real-time data synchronization capabilities using a Distributed Controller MVC Architecture (although synchronization happens only from the server to the client in this case, as the client does not maintain a Model).

High data volumes can also be handled by this architecture. Using techniques like lazy loading, as Twitter does, data can be loaded on demand. However, depending on the amount of data

and if the data should further be processed, an architecture with a client-side Model may be the better solution.

4.2.3 Synchronized Model MVC Architecture

The next step towards a Rich Client is to bring the Model into the browser. In opposition to the Controller in the architecture discussed before, there is no distribution of Model responsiblity across the network nodes. Instead, the client-side Model is a copy, or an excerpt, of the original, server-side Model. The one in the browser is the operating Model — all client-side manipulations are executed on the client-side Model. The server-side Model is used for persistence, which requires a synchronisation strategy between client and server.

The synchronisation strategy and possibilities of the client-side Model highly depend on what the Model represents.

- If the client-side Model is an *exact copy* of the server-side one (i.e. it keeps all the data that are also contained on the server-side), all Views and Controllers can directly interact with this exact Model. Synchronisation can happen
  - periodically, i.e. after a defined amount of time (“timeout”)
  - on special actions, e.g. manipulation of the client-side Model

  It is recommended to only synchronize the differences between the client and server, usually called “delta”.
- The client-side Model can also be only a *partial copy* of the server-side one. By keeping track of which part of the data is already present on the client, data can be efficiently transferred from and to the server. This is preferred for high volumes of data.

  The synchronisation strategies are the same as described above. Lazy loading can best be implemented using a partial client-side Model.

Structure

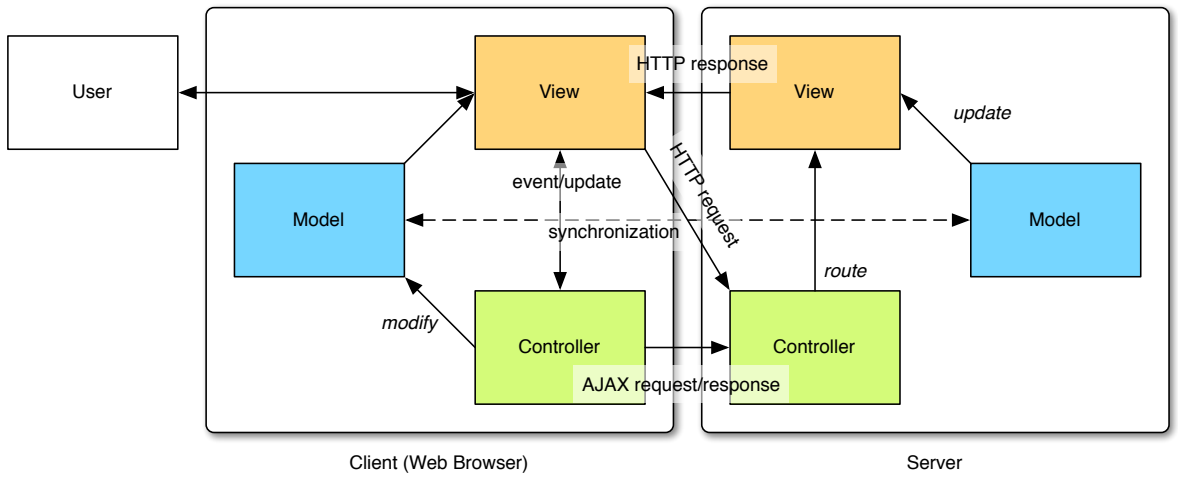

Figure 4.4: Structure of a Synchronized Model MVC Architecture

In this architectural diagram, the synchronization between the client-side and server-side Models is illustrated using a dashed line.

Implementations

Once Model synchronisation is part of the architecture, a client-side MV* framework is often used to structure the JavaScript code; this is especially useful, as basic functionality for synchronisation and data I/O do not have to be implemented manually, but are provided by most frameworks.

There are a lot of examples that make use of the Backbone.js framework. The open-source social network *Diaspora*[1], for example, uses Ruby on Rails as a server-side MVC framework, and in parallel Backbone.js on the client-side. Although Ruby on Rails follows MVC and Backbone.js is built after MVP, these two frameworks work together well.

Backbone.js Models can directly synchronize with Rails Models without any further transformations, as they can be configured to use the same JSON format for data exchange.

One can argue that the presence of a complete MV* triad on the client makes the server-side Controller obsolete: it is no longer needed to manipulate the server-side Model. This is correct

[1] See https://joindiaspora.com/

for completely client-sided MV* applications, as described in the next section (*Rich Client MVC*). Rich Client MVC applications are one-page applications, which means that their user interface and routing is handled completely using JavaScript. But in applications following the Synchronized Model architecture, both routing and initial creation of the user interface happens on the server-side.

Another advantage of the server-side Controller is the fact that multiple front-ends can be built to collaborate with the server. In addition to the web client, mobile clients that follow a thin client paradigm — for example for performance reasons — can make use of the completeness of server-side MVC.

Applicability

Criterion	Applicability
Data Model Complexity	*Depends*
Real-time Model Synchronization	*Yes*
Data Volume	*Yes*

Table 4.3: Applicability of a Synchronized Model MVC Architecture

The parallel structure of a Synchronized Model MVC Architecture makes the handling of comples data models difficult. Data can be changed on two different points in the application: on the client, using the client–side Controller, and on the server, using the client–side Controller to send an AJAX request to the server–side Controller, which then modifies the Model. Of course it is possible to efficiently resemble complex data models with this architecture variation, but with a lot of Models and cross-references included, it is possible that the structure becomes intransparent.

The synchronization of the Model makes applications with real–time snychronization possible, even for more complex data structures. Using the techniques described as and WebSockets on page 37, every synchronization strategy can be implemented, whether the client-side Model is a complete or only a partial copy of the server-side Model.

This architecture is suitable for processing high data volumes, for the same reason as it is suitable for applications that require real-time data synchronization. As the Model is synchronized

between client and server, it is possible to only load the required data onto the client-side and load additional data subsequently, on demand.

4.2.4 Rich Client MVC Architecture

The Rich Client MVC Architecture is described as "RIA Architecture"[1] by Steele (2004). It places the whole set of Model, View and Controller on the client-side, while only a Model stays on the server-side.

What distinguishes this variant from the *Synchronized Model MVC Architecture* is that there is no server-side View or Controller anymore. Of course, there is an entry point to the application needed. This is usually an almost empty HTML file that only contains a skeleton and references to additional resources, such as JavaScript, CSS and image files that have to be loaded subsequently. The user interface is being constructed on the client side, purely through JavaScript code.

Structure

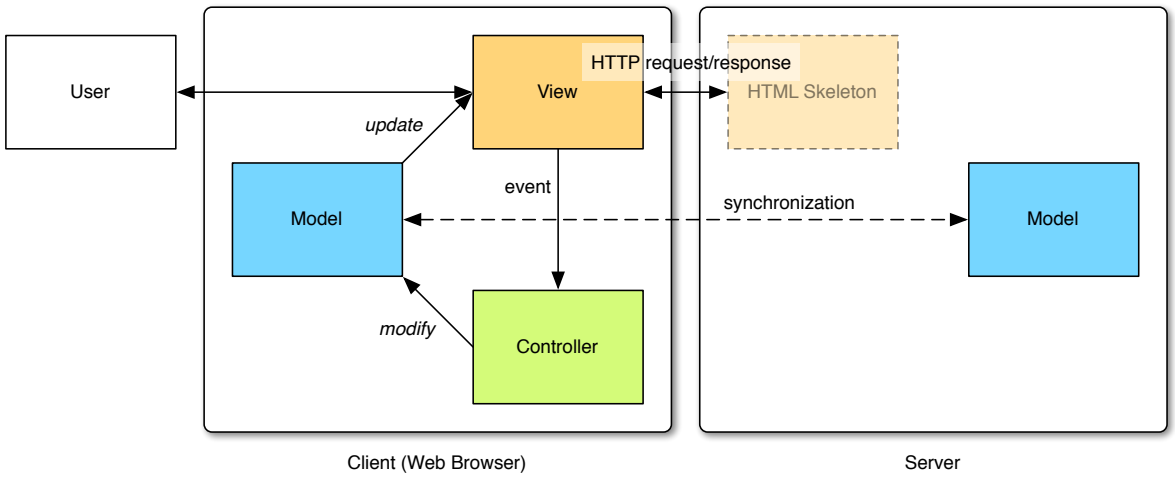

Figure 4.5: Structure of a Rich Client MVC Architecture

[1] RIA is short for *Rich Internet Application*. The term can be used for rich web clients, but it usually describes applications developed with Adobe Flash or Flex, see Section 3.2.1: *Terminology*

Implementations

The Dojo Toolkit[1] comes with a great variety of different modules and widgets (called *Dijits*). Dojo 1.6[2] makes it possible to build pure client-side MVC applications.

In Dojo, the Model is a Dojo Data Store, for example `dojo.store.Memory` and `dojo.data.ItemFileReadStore`. Using the `dojo.store.Observable` wrapper around the store, the Observer pattern can be established between View and Model. Views can be Dijits, either Dojo's own or custom ones. Controllers can be implemented as own modules, but often Controller code is put inside of the Dijit, which breaks with the *Separation of Concerns* paradigm of MVC.

Using the `dojox.data.JsonRestStore`, a RESTful web service can be connected to the client-side store, which allows data persistence on the REST server and makes a Rich Client/Thin Server configuration possible. Dojo's MVC implementation is further discussed in Chapter 5: *MVC in IBM Content Navigator*.

Applicability

Criterion	Applicability
Data Model Complexity	*Yes*
Real-time Model Synchronization	*Depends*
Data Volume	*Yes*

Table 4.4: Applicability of the Rich Client MVC Architecture

A web application with a Rich Client MVC Architecture is able to handle complex data models. The most important requirement for this is a well-defined interface between the client-side and server-side Models. As the code to process and consolidate Model data is situated on the client-side only, even very complex domain data can be handled by this application architecture.

The Rich Client MVC architecture is suitable for real-time model synchronization, as long as a powerful modern web browser is used. This requirement has two reasons:

[1] See http://dojotoolkit.org/

[2] Please note that Dojo developed various MVC approaches over time, and they differ quite a lot between versions 1.8 (the latest as of September 2012) and older ones. Dojo 1.7 provides the `dojox.mvc` package, but in this thesis, Dojo 1.6 is the version referred to, as it is the version used by IBM Content Navigator at this time.

- Depending on the complexity and the number of Models to synchronize, the *Comet* technique described on page 4.1.2 might not be feasible anymore. Too many iframes or pending requests in parallel can slow down the user interface. Using a modern browser that supports HTML5, WebSockets can be used for synchronization, which also allow more control over the data transfer.
- Due to the fact that the whole application is situated on the client-side, a lot of memory might be needed. Also, the execution time of JavaScript code can be slow if the JavaScript engine is not capable enough to run a full-fledged, desktop-level web application.

Regarding data volume, the same is true for Rich Client MVC as it is for a Synchronized Model MVC architecture.

4.2.5 Comparison and Conclusion

The results of the applicability evaluation are summarized in Table 4.5.

MVC Architecture	Data Model Complexity	Real-time Model Synchronization	Data Volume
Server Side	*Yes*	*No*	*No*
Distr. Controller	*No*	*Yes*	*Depends*
Synchr. Model	*Depends*	*Yes*	*Yes*
Rich Client	*Yes*	*Depends*	*Yes*

Table 4.5: Applicability of the different MVC architectures

Both the Server Side and Rich Client MVC architecture are characterized by their rather simple structure. This fact makes them preferable for applications with complex data models, as the data have to be modeled only on one of the two tiers. For applications that need to process data in real-time or large amounts of data, the three architectures based on AJAX are suitable. They allow loading of data at virtually any time during execution of the application and can use techniques like *lazy loading*.

5 MVC in IBM Content Navigator

This chapter introduces the IBM Content Navigator application and framework. After an overview on its architecture and plug–in system, the main client-side components Dojo and IDX are outlined. To conclude this chapter, the design and implementation of a log analysis plug–in is discussed.

5.1 Architecture and Extension Points

IBM Content Navigator is an enterprise web application written in Java and JavaScript. The server–side part is designed to run on a application server, the client–side part runs in a web browser. The architecture of Content Navigator, which is presented here, allows the extension through plug–ins at many points of the application.

5.1.1 Architecture: Layers and Components

IBM Content Navigator is designed after the *Separation of Concerns* principle. This is true for the different layers of Content Navigator on the one hand, and the different tiers it uses on the other hand. The layers of ICN are shown in Figure 5.1.

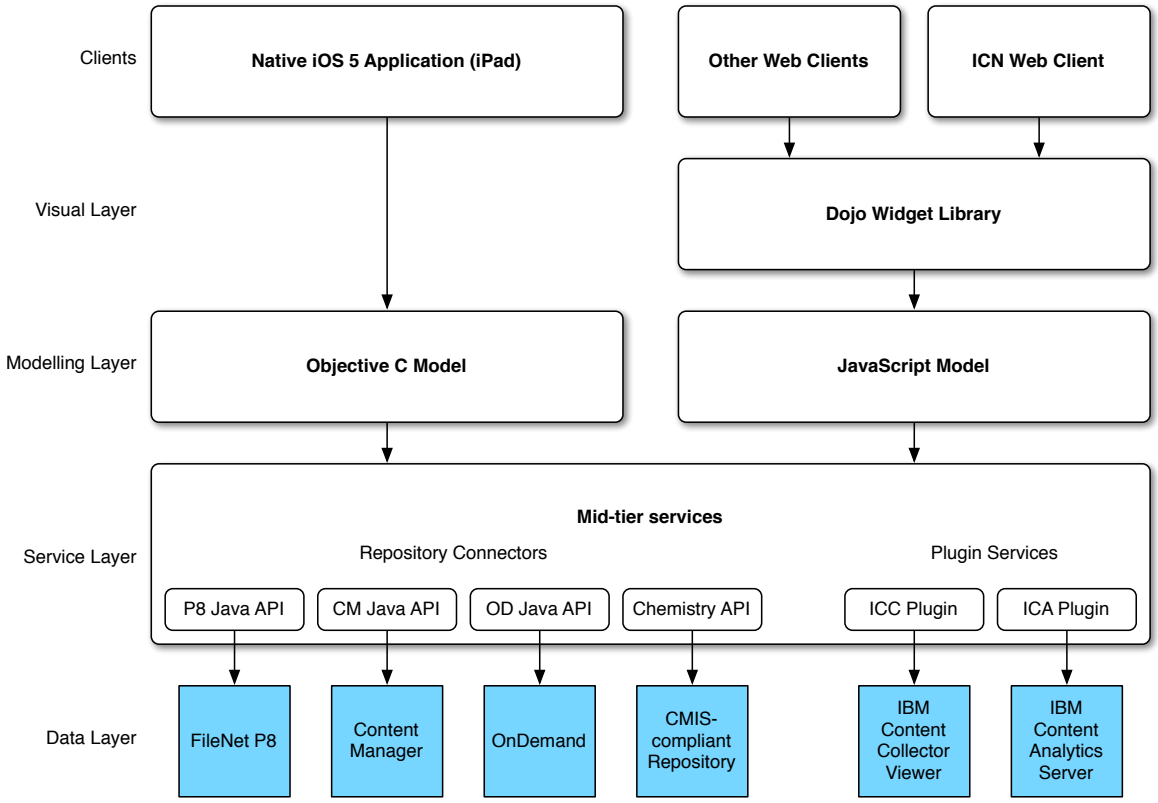

Figure 5.1: Layers of IBM Content Navigator

The top layer represents the actual clients for Content Navigator. Besides the desktop web client, there exists also a mobile iOS 5 client, which is not in the scope of this thesis. The client layer makes use of the visual layer underneath that contains all the Dojo widgets (*Dijits*). In turn, the widgets access the modelling layer, which contains JavaScript Models held in the browser. On the one hand, these models can be connected to mid-tier services, such as ECM repository APIs, on the other hand, they can be directly connected to external, HTTP-based services, such as REST.

The mid-tier services are implemented in Java and run on the application server, whereas the modelling and widget layers run in the browser. This is illustrated by Figure 5.2, which shows the tier architecture of Content Navigator.

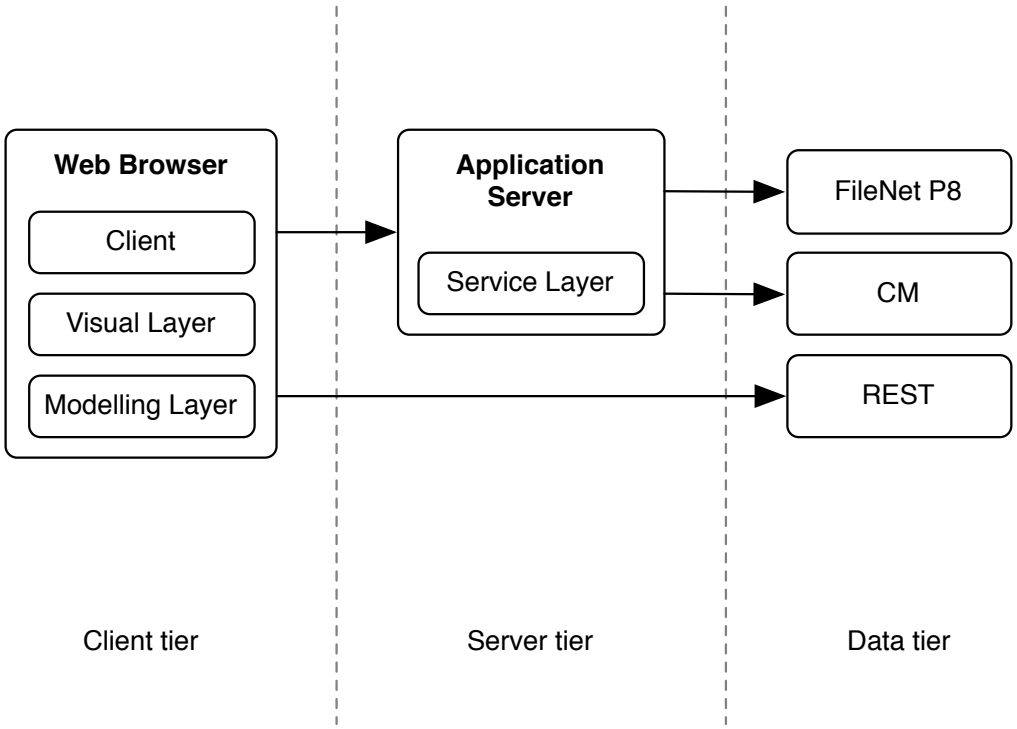

Figure 5.2: 3-Tier architecture of IBM Content Navigator

5.1.2 Extension Points: Plug–ins

The modular architecture previously presented makes it easy to extend Content Navigator on different levels. Plug–ins for ICN can include different components that can directly be used inside of ICN, as described by Zhu et al. (2012, pp. 103ff.). The ones relevant for the log analysis plug–in are *Plugins* themselves, *Features* and *Widgets*.

Plugins are the containers for the following extension points, which are registered with ICN through the "Plugin" Java class.

Actions are buttons or menu items that can be triggered by the user; they can be added to the existing UI of ICN.

Menus of the existing UI can also be created and customized, just like *Actions*.

Plug–in Services allow developers to extend the service layer of Content Navigator. They can implement arbitrary Java code, but can be called from the client-side JavaScript as they are exposed by a Servlet.

Using a *Service*, it is possible to build anything that Java is capable of. This can be, just to name a few examples, a proxy for REST calls, an additional connector to a content repository not already covered by ICN, or a system to save data on the server side. The IBM Content Analytics (ICA) plug–in uses *Services* to connect to an ICA server via SSO.

Features are areas of the application UI. In ICN, the Browser, Favorites, Team Spaces, Search, Work and Administration *Features* do already exist and can be accessed using the Feature Pane on the left side of the UI (when logged in).

Features are associated with a Dojo class to specify the View that should be shown when the *Feature* is accessed.

Viewers are used to display the content of a specific document type.

Layouts can customize the overall layout of the application UI, whereas *Features* can only define a certain application area. ICN's standard layout (Banner Bar at the top, Global Menu below, Feature Pane on the left and Features filling the remaining screen space) can be changed using *Layouts*.

Request and Response Filters are used to change the JSON sent from and received by *Services*.

Widgets are Dojo Dijits that can be included in Plug-ins. They can either be developed from scratch or extend already existing Dojo or IDX Dijits. More on Dijits is written in Section 5.2.

Besides these components, arbitrary JavaScript code, best created as Dojo modules, can be written and packaged into the plug–in. This includes Model and Controller code. Figure 5.3 illustrates the connection between extension points and the ICN layer architecture (please compare to Figure 5.1).

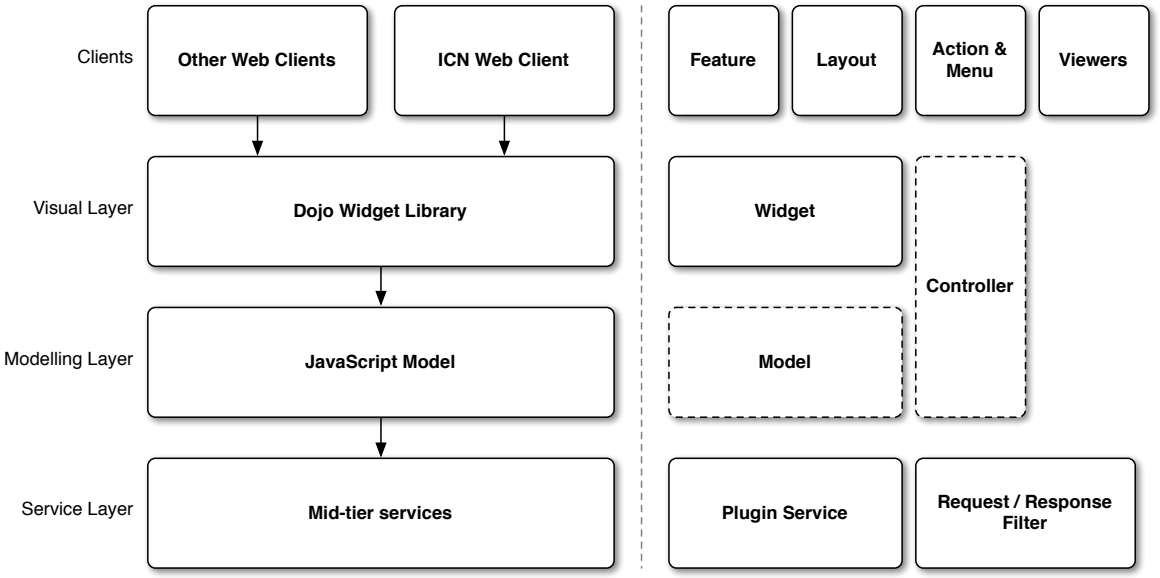

Figure 5.3: Plug–in components extending ICN

As can be seen, plug–ins cannot extend the data layer, as it is not part of IBM Content Navigator itself, but rather composed of external data sources, such as content repositories and REST services.

5.1.3 Deployment and Configuration

Content Navigator itself is packaged — without any plug–ins — into an Enterprise Archive (EAR) file and can directly be deployed on an application server, such as IBM WebSphere Application Server. The connection to a database for storing and loading the configuration is needed and can be set up using the ICN initialization tool.

Plug–ins are deployed separately. They are packaged as Java Archive (JAR) files and placed at any location that is accessable via a URL. Plug–ins can be loaded using the administration pane of ICN. This separation between the actual application and its plug–ins allows the application administrator to update, load and unload plug–ins without restarting ICN or the application server.

Different ICN-based applications, called *Desktops*, can run on the same ICN installation. A Desktop can be configured using the ICN administration pane; it is assigned a *Layout* and a number of *Features* (as described in Section 5.1.2), as well as content repositories and other options.

A Desktop is chosen via the URL, for example `http://localhost/navigator?desktop=sccm`. Figure 5.4 illustrates how applications can be assembled using *Features* of different plug–ins.

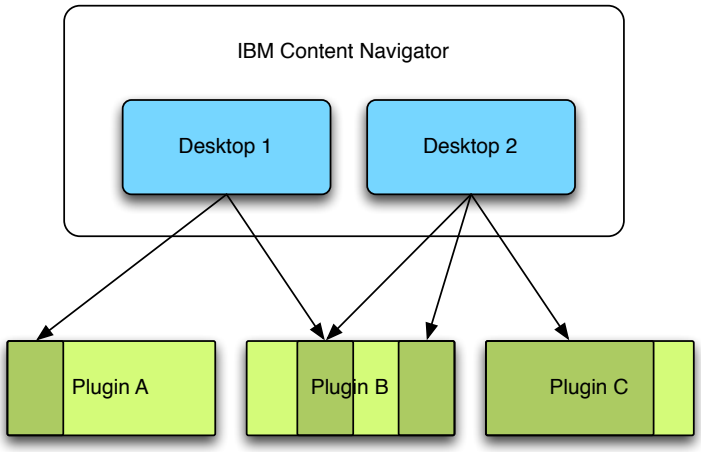

Figure 5.4: ICN Applications using different Plug–in Features

5.2 Dojo Model–View–Controller

This section introduces the components of Dojo that are relevant to building Model–View–Controller applications, and those that were especially used while building the log analysis plug–in for SCCM. As IBM Content Navigator 2.0 (which is the latest version as of September 2012) includes Dojo 1.6, this is the version of Dojo referred to in this section.

5.2.1 Dojo Basics

The Dojo Toolkit, usually just called *Dojo*, is a set of libraries to build JavaScript applications. In opposite to other JavaScript libraries, such as jQuery, Prototype or Mootools, Dojo is not only aimed at simplifying DOM manipulation, AJAX and event handling, but also comes with an extensive widget library (*Dijit*) and graphics/charts API.

Dojo is separated into three packages:

Dojo is the core package and contains basic functionality, such as DOM manipulation, AJAX, effects, and JavaScript language helpers. Its is `dojo`.

Dijit contains all stable widgets, such as layout widgets, forms, dialogs, and a tree view. Its namespace is `dijit`.

DojoX are the Dojo extensions in the namespace `dojox`. This package contains less common components, such as the graphing and charting API. The DataGrid (`dojox.grid.DataGrid`) and JsonRestStore (`dojox.data.JsonRestStore`), which are part of DojoX, are further discussed in Section 5.3.

Classes and Objects

JavaScript is a prototype-based object-oriented programming language. Dojo simulates class-based object orientation, which is more familiar to developers used to Java, C++, C# and other languages. To illustrate this, Listing 5.1 contains the declaration of a Dojo class using `dojo.declare`.

```
1 dojo.declare("my.Thinger", null, {
2   constructor: function(/* Object */args){
3     dojo.safeMixin(this, args);
4   }
5 });
```

Listing 5.1: Declaring a Dojo class

This listing declares the `my.Thinger` class (in the `my` namespace). When instantiating an object, the constructor mixes the `args` argument, which should be a JavaScript object, into the new instance. The result of this mixin is shown in Listing 5.2.

```
1 var thing = new my.Thinger({ count:100 });
2 console.log(thing.count);
```

Listing 5.2: Instantiating an object using a Dojo class

The object[1] provided as an argument to the `my.Thinger()` constructor is mixed into the new object `thing`. This means, that `thing` now contains the properties of this object, which can be proven by printing out `thing.count` in the console.

[1] JavaScript objects can simply be created as literals. Object literals are enclosed in braces ('{' and '}') and contain a comma-separated list of properties (attributes and methods).

All Dojo components, including Dijits, are created using this concept, and so are the classes developed for the log analysis plug–in.

Scope

Another of Dojo's concepts being of help in MVC applications and the plug–in is `dojo.hitch`. The JavaScript API of web browsers features a callback-based event system. This means that callbacks are registered to an event and triggered when the event occurs. A frequently faced problem is that the function has a different scope than the developer would expect (Flanagan 2006, pp. 53–56 and pp. 180–185). In JavaScript, this is typically solved using a *closure*, a concept to pass on a given scope to another function. In applications with a lot of callbacks, which may also be depending on each other to execute, closures can have a negative impact on the code maintainability and comprehensibility. `dojo.hitch` returns a function that, when executed, has a specified scope, as shown in Listing 5.3.

```
function myCallback() {
  console.log(this.localVariable)
}

dojo.declare("my.Thinger", null, {

  localVariable: "I am in the Thinger scope",
  button: null,

  constructor: function() {
    this.button = new dijit.form.Button({
      label: "Hello"
    });
    dojo.connect(this.button, 'onClick', dojo.hitch(this, myCallback));
  }
});

var thing = new my.Thinger();
```

Listing 5.3: Instantiating an object using a Dojo class

This listing first defines a function that acts as a callback and refers a variable ("localVariable") of its own scope ("this"). The Dojo class declared below introduces said variable and

constructs a button. In line 14, the "onClick" event is registered with the callback through `dojo.hitch(this, myCallback)`. It means that the scope of `myCallback` at execution time of the callback is set to what `this` has been in line 14.

5.2.2 MVC Components

Dojo provides different components that make it possible to build applications using a MVC architecture. In opposition to JavaScript frameworks that were built towards *structuring* an application using MVC (or a comparable pattern), such as Backbone.js, Ember.js and JavaScriptMVC, Dojo lets the software engineer decide on the architecture.

Model

The modules `dojo.data` and `dojox.data` provide the data modelling layer. They contain a number of Model equivalents, called *Stores*, all of which implement one or more interfaces to handle data (Zyp n.d.):

- The *Read API* forces the Store to provide methods and maintain internal structures to expose a Store's data
- The *Write API* enables a Store to accept data manipulations
- The *Identity API* forces the Store to manage an identifier for each single data item. It must be possible to look up a data item using this unique identifier.
- If implementing the *Notification API*, other objects can connect to events that fire when data in the Store are changed (an implementation of the *Observer* pattern).

Dojo already implements various stores that serve different purposes and can use a specific data source. The `ItemFileReadStore`, for example, takes a file of JSON data as the data source (specified by a URL). It implements the *Identity* and *Read* APIs, so data can be read, but not written. As the source is a file and data cannot be manipulated, it is not necessary for this store to implement the *Notification* API.

The `JsonRestStore` serves a more complex purpose. It is connected to a REST service and thus needs to implement both the *Read* and *Write* APIs. But the other two APIs are implemented also — this is especially important as this store is often used in cooperation with a Dijit, such as the

Tree or DataGrid. To keep the Dijits up to date, a notification mechanism as provided by the *Notification API* is necessary (see next section for more information on the Tree and DataGrid).

View

In Dojo, the View usually is a single Dijit or a collection of interrelated Dijits. This section presents two data-centric Dijits that are part of Dojo 1.6.

The *Tree* (`dijit.Tree`) is a Dijit to display hierarchical data. It cannot connect directly to a Dojo store, but needs an additional API to handle the nested data model. These API can either be implemented manually by mixing in the according functions into the store when instantiating it, or the store can be wrapped by either one of the already existing `TreeStoreModel` and `ForestStoreModel` . The first expects a single root item, whereas the second expects multiple root items (thus the name *Forest*).

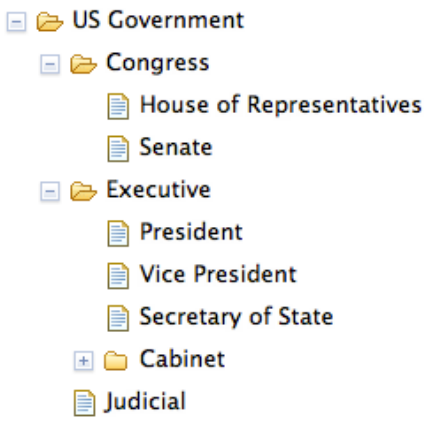

Figure 5.5: Screenshot of a Dojo Tree

Listing 5.4 shows the creation of a store, a TreeStoreModel wrapper around that store and a Tree connecting to the store via the TreeStoreModel. A REST service at the URL `data/` is presumed, and the data returned must have a special structure: the attribute "name" has to be present in every item, as it is used as the label, and an array with child items has to be placed unter the "children" attribute.

Listing 5.5 shows a sample JSON structure that is returned by the REST service at `data/` .

```
dojo.require("dijit.Tree");
dojo.require("dojox.data.JsonRestStore");
dojo.require("dijit.tree.TreeStoreModel");

store = new dojox.data.JsonRestStore({
  target: "data/"             // URL to the REST service
});

treeModel = new dijit.tree.TreeStoreModel({
  store: store,               // the store to wrap around
  query: {tree:"root"},       // root item
  labelAttr: "name",          // name of the label attribute
  childrenAttrs: ["children"] // attributes holding children
});

tree = new dijit.Tree({       // create a tree
  model: treeModel            // give it our model
}, "tree");                   // target HTML element's id

tree.startup();
```

Listing 5.4: Creating a Tree with a store and TreeStoreModel

```
{
  name: "World",
  children: [
    { name: "Africa" },
    { name: "Australia" },
    { name: "Asia" },
    { name: "Europe" },
    { name: "North America",
      children: [
        { name: "Canada" },
        { name: "USA" }
      ]
    },
    { name: "South America" }
  ]
}
```

Listing 5.5: Sample JSON data for the Tree Dijit

The *DataGrid* (`dojox.grid.DataGrid`) is a Dijit to display tabular data, similar to a spreadsheet. It can be used to display high amounts of data, using features like editable cells, custom cells that contain Dijits or arbitrary HTML code, and lazy loading. Its extension `dojox.grid.TreeGrid` can even display hierarchical tabular data using expanders and aggregates.

In opposition to the Tree, the DataGrid can directly access the data of a store. That store has to implement at least the *Read* and *Identity* APIs. The DataGrid also needs a *structure*, which is an object describing the cells of the grid and the source of the cells' data. In the simplest case, a cell can contain the value of an attribute of the according item, but the cell content can also be custom, for example a Dijit or an image.

As the DataGrid is one of the most frequently used Dijits in the log analysis plug–in that was developed in the course of this thesis, an extensive example of implementing it is given later in Section 5.3.3.

In Dojo, a View can also be completely built by hand. An example for that is shown in Section 5.3.4: a simple, data-displaying IDX Dijit is taken and methods are added to connect it to a Dojo store.

Controller

For the Controller, Dojo does not provide any special classes or APIs. The Controller code can be any code that binds callbacks to View events and performs updates on one or more Models.

An example of a Controller is given in Section 5.3.4, which shows the implementation of a custom Model–View–Controller triad using Dojo and the IDX Dijit `idx.grid.PropertyGrid` .

5.3 Implementation of a Log Analysis Plug–in

In the context of this Bachelor's thesis, a log analysis system was developed by two students. The author created a front-end using IBM Content Navigator, whereas Metzger (2012) implemented the respective back-end using IBM BigInsights.

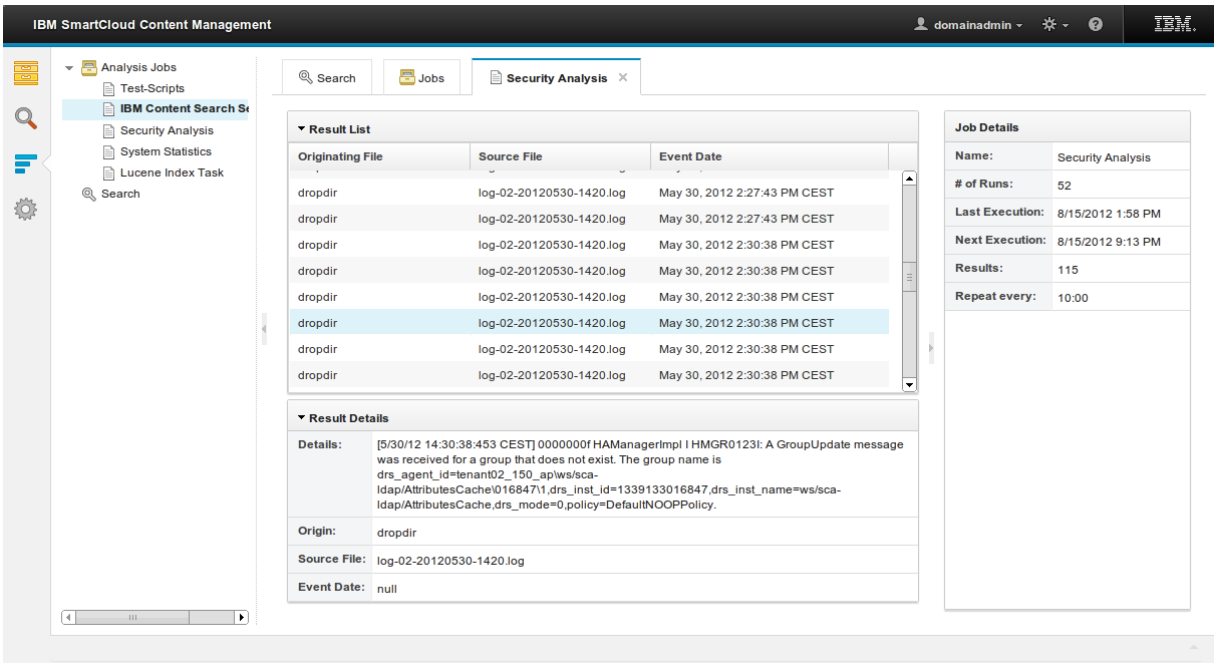

Figure 5.6: Screenshot of the log analysis plug–in UI

5.3.1 Motivation

As part of IBM's ECM portfolio, SmartCloud Content Management is a Software–as–a–Service for document and email compliance archiving. It is a large integration project that makes use of many IBM software products, such as DB2[1], FileNet P8, WebSphere Application Server and eDiscovery Manager[2], to provide an out–of–the–box archiving solution to customers. In this cloud environment, massive amounts of operational data are generated in the form of log files, especially if multiple customers are involved.

As IBM is responsible for the operation of SCCM, problems have to be solved immediately as they arise, and steps must be taken to prevent problems beforehand. The large amounts of log

1 IBM DB2 is a relational database management system (DBMS).

2 IBM is an ECM product that finds documents related to a specific business case, for example a legal case, in a content repository.

data that the integrated software products generate contain knowledge that is necessary for *predictive analysis*. Using IBM BigInsights, correlated log entries can be found and integrated to gain insights on the system's state and arising problems.

These analysis results have to be made available to the cloud operator in an easily accessable manner. Using Content Navigator to develop a front-end for the log analysis system not only makes it possible to integrate it with any ICN based product, but also emphasizes its ability to serve as a flexible, versatile web application framework.

The design and implementation of the BigInsights back-end are not in the scope of this thesis, as they mostly reflect the work of Metzger (2012). Nevertheless the interface to the back-end — a REST service — is presented, as it is essential to some functionality of the front-end.

5.3.2 The REST service

To create a log analysis plug–in and the according REST service, a basic understanding of the data being collected and shown must be present.

Robert Metzger built a data aggregating system using IBM , which is a suite of different products related to analysis. Metzger used different tools out of this suite to create two services: a full-text search for log files and a framework for data aggregation *jobs*. These jobs can be defined using a powerful script language. They run periodically to collect log files, aggregate related log file entries and extract single information out of each collected entry.

Both these services are accessible via a REST service written in Java. The results of both are generated by the BigInsights system and are not meant to be manipulated or deleted by a user. For this reason, the REST service is read–only by design. Strictly spoken, this implementation of REST is incomplete, as there is no *POST*, *PUT* and *DELETE* capability provided.

Data format

As the data format for the REST service, JSON was chosen. On the one hand, JSON is flexible without having a lot of overhead (compared to XML), on the other hand, it is natively supported by both the BigInsights products used and JavaScript (and Dojo, respectively). This makes it the preferable data format for this application scenario.

Services

The full-text search is available at the REST server's address at the URL `/search`. To initiate a query, the GET parameter "query" has to be provided with the according query string. A complete query for the string "security" looks like this: `127.0.0.1/search?query=security`, assuming the REST service runs on localhost[1].

As a result, the search service sends a JSON array containing the found log entries as objects. Every object contains the source (the machine and service that wrote the log file), the entry's and the exact log entry content.

The analysis jobs are accessed differently. The URL `/jobs` returns a list of all jobs that are known to the system. Besides other attributes, every job has a unique identifier which can be used to further access the job results via `/jobs/%identifier%`, where `%identifier%` has to be replaced with the according job identifier.

The list of jobs includes a description of every job. This description consists of metadata, such as the total number of collected results, as well as the structure of a result item. As every job can return completely different data, the client has to know the data structure, the type of every field (e.g. "date" or "integer"), and if the field should be displayed or not. An example of a job description is given in the Appendix on page **??**.

Advanced features

The REST service implements sorting and lazy loading. Both features are built to complement the client-side REST implementation of the `dojox.data.JsonRestStore`.

Lazy loading is a form of pagination. But instead of returning a certain "page" of data, the REST service returns a "range". This range is specified by GET parameters. The store on the client side keeps track of which items are already loaded. It then can request only those that are not loaded, but are needed to be displayed next.

Sorting is also implemented in the REST service on the server side. This may seem inconsequent in a Rich Client application, but it is necessary. Sorting can only be performed if all the data to be sorted are available. As the jobs can return more than 10.000 results, and not all results are loaded on the client due to lazy loading, it is unlikely that the client-side Model contains *all* the

[1] In IP networks, the IP `127.0.0.1` always refers to the `localhost`, the very one host that sends the request.

items. Because the client would not be able to sort result items correctly if some are missing, sorting is implemented on the server instead.

5.3.3 Implementation

This section gives a high-level overview on the architecture of the plug–in, especially on the MVC components. Exemplarily, one set of MVC components is presented in more detail.

The log analysis plug–in integrates into IBM Content Navigator and connects to the previously described REST service. The high-level architecture is shown in Figure 5.7.

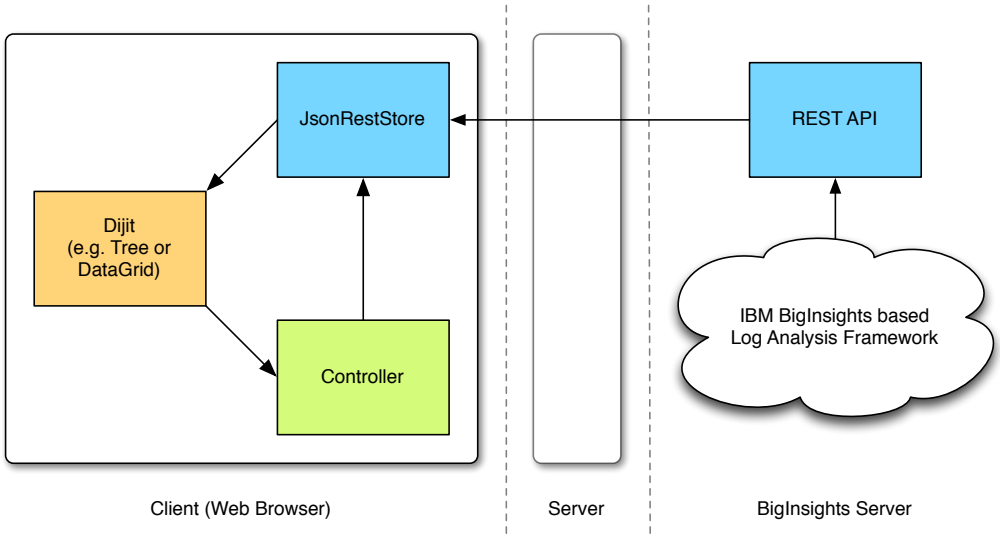

Figure 5.7: Integration of the log analysis plug–in into Content Navigator

File Structure

The plug–in's JAR file contains a folder hierarchy, which on the top levels represents the Java package of the plug–ins. The path `/com/ibm/ecm/extension/sccm/` maps to the Java package `com.ibm.ecm.extension.sccm`. It contains two classes, the `SCCMPlugin` class, which represents the plug–in object within ICN and contains information on the plug–in components, and the `LogAnalysisFeature` class, which represents the *Feature* that is exposed through the plug–in.

One level deeper, the `WebContent` folder contains the main CSS and JavaScript files of the plug–in. `sccmPlugin.css` contains custom styles needed in the plug–in, while `sccmPlugin.js` loads the required Dojo classes using `dojo.require()` statements.

The folder hierarchy below `WebContent` can partially be mapped to JavaScript namespaces, but also contains additional resources:

`sccm/layout` contains Dijits used for layouting in the `sccm.layout` namespace

`sccm/model` contains Models in the `sccm.model` namespace

`sccm/view` contains Dijits used as Views in the `sccm.view` namespace

`sccm/resources` contains images and CSS files

The Dijits developed for the log analysis plug–in are based on , so every folder that contains Dijits also contains a `templates` folder with the according HTML files.

`sccm/ConfigurationPane.js` is a layout Dijit that gets included in the administration panel of ICN. It can be used to provide configuration options for the plug–in, such as the host and port of the log analysis REST service.

User Interface Integration and Layout

To access the functionality of the log analysis plug–in, a new *Feature* was created. When integrated into a Desktop, this Feature can be accessed using the Feature pane on the left side of the ICN interface.

The layout of the log analysis Feature was created using *container Dijits*. These are Dijits that can themselves hold other Dijits, such as other container Dijits or Dijits that act as Views (i.e., display data).

`sccm.layout.LayoutPane` is the basic layout Dijit of the log analysis *Feature*. It contains a `idx.layout.BorderContainer`, which divides the screen area into a left and right area. The left one is filled with a `dijit.Tree` to display all elements of the plug–in: the jobs overview, the full–text search and all existing jobs. Using this tree, the user can open the according Views in a tab. The right area of the LayoutPane holds a `dijit.layout.TabContainer` to display *tabs*. Two tabs are initially included, containing the layout Dijits `sccm.layout.Search` and `sccm.layout.Overview`.

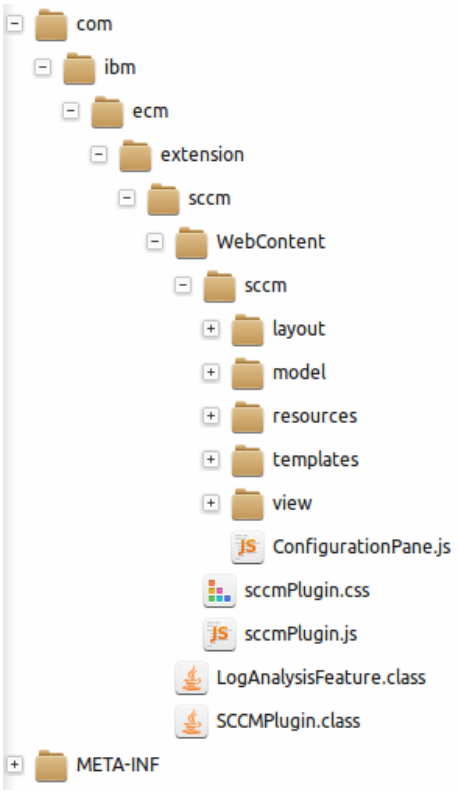

Figure 5.8: Folder hierarchy of the log analysis plug–in

`sccm.layout.Search` is a layout Dijit containing Views to perform the full–text search. It is instantiated inside the TabContainer of `sccm.layout.LayoutPane`, but could be used anywhere. The Search Dijit contains an input field and button to enter and submit the search term, as well as a `dojox.grid.DataGrid` to display the search results.

`sccm.layout.Overview` contains a `sccm.view.StatsView` Dijit to display information on the state of the log analysis system, and a `sccm.view.JobsView` to list the Jobs that are registered with the system. The Overview Dijit is meant to be displayed inside a Tab in the TabContainer of `sccm.layout.LayoutPane`.

`sccm.layout.Messages` is the only layout Dijit that is meant to be instantiated multiple times. It is used to display the details and results of a Job. For this reason, it contains a `sccm.view.DetailsView` for the Job details, as well as a `sccm.view.ResultsView` to list the results. A second `idx.grid.PropertyGrid` is used to display details for the selected result item.

For every Job, there is one instance of `sccm.layout.Messages` opened as a tab, if the user double clicks the according job in the Job Overview tab or in the navigational tree on the left side of the *Feature*.

Models

In the log analysis plug–in, all Models (or Dojo stores, respectively) are created as Singletons[1]. This prevents access to the server-side Model from two different points in the application, which could cause memory leaks or lead to refetching of already fetched data.

Additionally to the data stores already provided by Dojo 1.6, it would have also been possible to create a custom one. The decision for `dojox.data.JsonRestStore` is explained below:

ItemFileReadStore The simple, read-only store `dojo.data.ItemFileReadStore` can use JSON as input, but expects a strict format[2], which is not feasible for the use case of a versatile REST API.

JsonRestStore The `dojox.data.JsonRestStore` can be connected to a REST store and supports all CRUD[3] actions through HTTP methods. Although the REST API does not have

[1] The Singleton is a design pattern which ensures that only one instance of an object exists at a given time. It is described by Gamma et al. (1995, pp. 127–134).

[2] See http://dojotoolkit.org/reference-guide/1.6/dojo/data/ItemFileReadStore.html

[3] *Create*, *Read*, *Update*, *Delete*

write capabilities, the JsonRestStore is the best match for the log analysis Models. All features that are required by our use case (and more) are supported by this store, for example the ability to request only a part of the data that is not already loaded; this is used to implement *lazy loading*.

The namespace `sccm.model` contains all Models. The Model to manage the Job descriptions is instantiated as `sccm.model.Jobs`, whereas the search Model is created as `sccm.model.Search`.

The other Models, which hold the Job results, are instantiated when they are used for the first time. The object `sccm.model.messages` contains a method to initialize Models, which is shown in Listing 5.6. All Models of job results are managed as attributes of this object, identified by their unique job name.

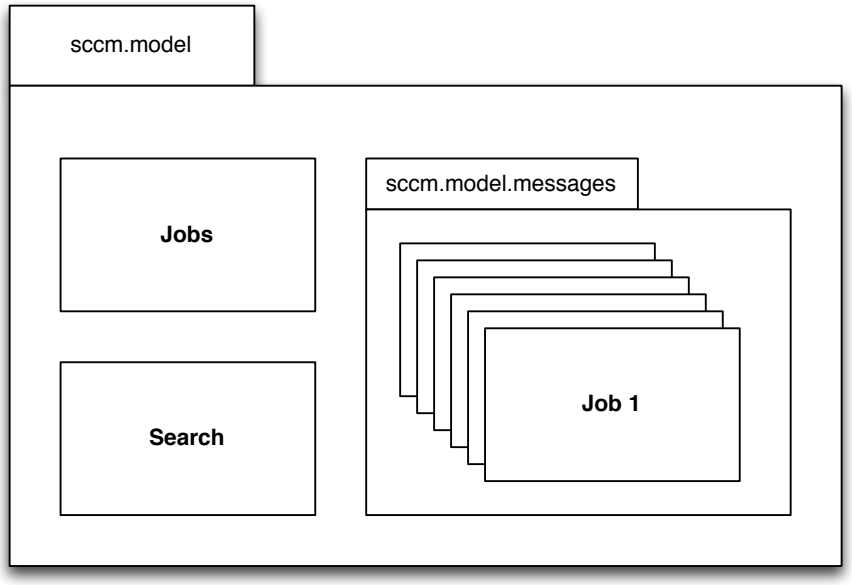

Figure 5.9: Plug–in Models in the according namespaces

```
sccm.model.messages = {
  getModel: function (name) {
    if(this[name] == undefined) {
      this[name] = new dojox.data.JsonRestStore({
        target: "http://9.152.130.251:8182/jobs/" + name
      });
    }
    return this[name];
  }
};
```

Listing 5.6: Models for job results are created as Singletons

An exemplary initialization of a results Model for the Job "ibm-css" is shown in Listing 5.7. If the Model was not initialized before, it gets created then; otherwise, the reference to the existing Model is used.

```
theModel = sccm.model.messages.getModel('ibm-css');
```

Listing 5.7: Initialization of a Model

Views and Controllers

In the log-analysis plug–in, there is no View built completely from scratch. Although Dijits are *visual* Dojo modules, they contain View code as well Controller code. These parts are hard to distinguish sometimes, as Dojo 1.6 does not provide a clearly structured MV* architecture.

As described in Section 5.2, the actual Views are Dijits such as `dijit.Tree` and `dojox.grid.DataGrid`. The log analysis plug–in does not extend or enhance them, it just *uses* them.

The modules defined under the namespace `sccm.view` are not only Views, but View–Controller pairs. The Dijits, for example `sccm.view.StatsView`, are the Controllers, whereas the Views are defined inside their templates (or, programatically, inside the JavaScript code). In other words, the Controller *wraps around* the View. This is the common way to define Dijits in IBM Content Navigator, as can be seen in the *ICA Plug–in*, for example.

`sccm.view.NavTree` contains a `dijit.Tree` that displays all the plug–in's components. The search, jobs overview and job results can be accessed using this tree. The Controller code of this Dijit connects the Tree with the `sccm.model.Jobs` model

`sccm.view.StatsView` is a Dijit containing an `idx.grid.PropertyGrid` (see Section 5.3.4 for more information on the PropertyGrid). It is used to the status of the log analysis system.

`sccm.view.JobsView` contains a DataGrid to display the various log analysis jobs.

`sccm.view.ResultsView` contains a DataGrid to display an overview of the job results.

`sccm.view.DetailsView` contains a PropertyGrid to display details on a job result. This View is further described in Section 5.3.4

5.3.4 Implementation of a custom Model–View–Controller triad

Part of the IDX distribution is a Dijit called *PropertyGrid*. While the *DataGrid* provided by stock Dojo is a good solution to get an overview on a large set of data, the *PropertyGrid* is better suited to get a detailed view on a single object and its properties. In the log analysis plug–in, this was used in two places:

- To display the properties of a *Job* next to the list of results it returned,
- to display the system status and
- to display the details of the selected result item.

The last case is covered in this example. For every job, a *tab* is created in a TabbedContainer Dijit (as described in Section 5.3.3) using the `sccm.layout.Messages` Dijit. The tab contains a DataGrid to display an overview of all the result items `sccm.view.ResultsView`, as well as a PropertyGrid to display details on the item that is *selected* in the DataGrid `sccm.view.DetailsView`. Thus, the content of the PropertyGrid changes whenever a different item is selected in the DataGrid.

Name:	Security Analysis
# of Runs:	52
Last Execution:	8/15/2012 1:58 PM
Next Execution:	8/15/2012 9:13 PM
Results:	115
Repeat every:	10:00

Figure 5.10: The IDX PropertyGrid, displaying job details

Although the PropertyGrid is a data–driven Dijit, it is not designed to work together with a Dojo Store. The following example shows how this custom connection is established using the PropertyGrid as a View, a JsonRestStore through the DataGrid as a Model, and implementing Controller code to change the View's contents according to user actions.

One can argue that this is not a classic MVC construction, as there is no Observer relationship between the View (*PropertyGrid*) and the Model (the according *JsonRestStore*). This is correct, but as the Model is read-only, the Observer is not necessary. It would be simple, though, to add it subsequently, if the PropertyGrid is used in a different scenario or the Model becomes writeable.

Model

The Model in this example is a simple `JsonRestStore`, as defined by Listing 5.8, but it is not used by the View directly. Instead, its data are accessed through the DataGrid, which is backed by the `JsonRestStore`. The listing defines a store for the "ibm-css" job. The `dataStructure` object is assumed to contain the structure for the DataGrid, which is not important here.

```
1 dataStore = new dojox.data.JsonRestStore({
2   target: "http://localhost:8080/jobs/ibm-css"
3 });
4
5 dataGrid = new dojox.grid.DataGrid({
6   store: dataStore,
7   structure: dataStructure,
8 });
```

Listing 5.8: Model and DataGrid for the PropertyGrid

View

The PropertyGrid is created declaratively inside the template file of the tab.

```
<div
  dojoType="idx.grid.PropertyGrid"
  data="{}"
  properties=""
  dojoAttachPoint="detailsGrid"
></div>
```

Listing 5.9: Declarative creation of the PropertyGrid

Usually, the `data` attribute contains the data to be displayed, and the `properties` attribute contains information on how to format and label these data. In this case, both attributes are empty, as the information are set individually depending on the Model data.

Controller

As the Controller accesses two Dijits, it is placed one level higher, inside the `Messages.js` file of the `sccm.layout.Messages` Dijit. The code in Listing 5.10 is a simplified excerpt. It does not show the whole Dojo class developed for the tab of Job results, but only the parts relevant for the PropertyGrid. This code is also not in the context of the according Dojo class, but is extracted from it.

The code assumes a few objects to be present: `detailsGrid` is a reference to the PropertyGrid defined in Listing 5.9, `dataGrid` is a reference to the DataGrid, `job` contains the job description object (see Appendix and below).

The `structureForDetails()` function takes a *result structure* object as the parameter. This object is part of the JSON description of a job; it describes the structure of a result item that is returned by an analysis job. In the Appendix, the JSON format of a job description is listed — the `resultStructure` object, as part of it, is the object that gets passed in as a parameter here. `structureForDetails()` returns a string that lists all the properties that should be displayed in the PropertyGrid, and adds type information so that the properties are formatted properly. The returned string is in the format expected by the PropertyGrid for its `properties` attribute, as illustrated by Listing 5.9.

`detailsGrid` is a reference to the PropertyGrid Dijit. Using the `set()` method, the PropertyGrid's `properties` attribute is overwritten with a string created from the *result structure* object, as described above.

The `dojo.connect()` function call in line 26 binds a callback, the `onRowClick()` function, to the `onRowClick` event of the DataGrid. This means, the callback is triggered when a row in the DataGrid is selected.

`onRowClick()` receives the selected result item from the DataGrid and sets it as the `data` attribute in the PropertyGrid. It then uses the `refreshPropertyGrid()` function to refresh the respective DOM nodes.

```
function structureForDetails(structure) {
  var a = "";
  for (var item in structure) {
    if(item != "__parent" && item != "__id") {
      var s = "";
      s = item;
      if(structure[item].type == "string")
        s += "(str)";
      else if(structure[item].type == "integer")
        s += "(int)";
      if(structure[item].type == "timestamp")
        s += "(dtm)";
      a += s + ",";
    }
  }
  // Remove trailing comma
  return a.substring(0,a.length - 1);
}

function onRowClick(event) {
  var item = dataGrid.getItem(event.rowIndex);
  detailsGrid.data = item;
  refreshPropertyGrid(this.detailsGrid);
}

function refreshPropertyGrid (pg) {
  for(var i = 0; i < pg._rows.length; i++)
    pg._rows[i].reformat();
}

// Set properties according to result structure
detailsGrid.set(
  "properties",
  structureForDetails(this.job.resultStructure)
);

dojo.connect(
  dataGrid,
  "onRowClick",
  dojo.hitch(this, onRowClick)
);
```

Listing 5.10: Controller code for the PropertyGrid

6 Conclusion & Outlook

6.1 The Unhosted Movement

6.2 TodoMVC

Bibliography

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. & Angel, S. (1977), *A Pattern Language: Towns, Buildings, Construction*, Addison-Wesley, Oxford.

Eich, B. (2008), 'Popularity'. Accessed: 25/05/2012.
URL: *http://brendaneich.com/2008/04/popularity/*

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. & Berners-Lee, T. (1991), 'Hypertext Transfer Protocol – HTTP/1.1'. IETF RFC2616.
URL: *http://www.ietf.org/rfc/rfc2616.txt*

Flanagan, D. (2006), *JavaScript: The Definitive Guide*, 5th edn, O'Reilly & Associates.

Fowler, M. (2002), *Patterns of Enterprise Application Architecture*, 1st edn, Addison-Wesley Professional.

Fowler, M. (2006*a*), 'GUI Architectures'. Accessed: 08/03/2012.
URL: *http://martinfowler.com/eaaDev/uiArchs.html*

Fowler, M. (2006*b*), 'Passive View'. Accessed: 08/03/2012.
URL: *http://martinfowler.com/eaaDev/PassiveScreen.html*

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, 5th edn, Addison-Wesley, Massachusetts.

Johnson, R. (2003), *J2EE Design and Development*, Wiley Publishing, Inc., Indianapolis.

Metzger, R. (2012), 'Implementing a Scalable, Distributed Search and Data Analytics Engine for Log File Analysis into a Cloud based Archive Using Apache Hadoop'.

Osmani, A. (2012), *Learning JavaScript Design Patterns*, O'Reilly.

Reenskaug, T. (1979*a*), 'MODELS–VIEWS–CONTROLLERS'.

Reenskaug, T. (1979*b*), 'THING–MODEL–VIEW–EDITOR, an Example from a planningsystem'.

Reenskaug, T. (2003), 'The Model-View-Controller (MVC). Its Past and Present'.

Reenskaug, T. (n.d.), 'Trygve/MVC'. Accessed: 07/14/2012.
URL: *http://heim.ifi.uio.no/trygver/themes/mvc/mvc-index.html*

Schiemann, D. (2007*a*), 'The forever-frame technique'. Accessed: 07/27/2012.
URL: *http://cometdaily.com/2007/11/05/the-forever-frame-technique/*

Schiemann, D. (2007*b*), 'The long-polling technique'. Accessed: 07/27/2012.
URL: *http://cometdaily.com/2007/11/15/the-long-polling-technique/*

Selvitelle, B. (2010), 'The Tech Behind the New Twitter.com'. Accessed: 08/25/2012.
URL: *http://engineering.twitter.com/2010/09/tech-behind-new-twittercom.html*

Steele, O. (2004), 'Web MVC'. Accessed: 07/22/2012.
URL: *http://osteele.com/archives/2004/08/web-mvc*

Venners, B. (2009), 'Twitter on Scala'. Accessed: 08/25/2012.
URL: *http://www.artima.com/scalazine/articles/twitter_on_scala.html*

Wikipedia (n.d.*a*), 'Enterprise Content Management'. Accessed: 08/18/2012.
URL: *http://en.wikipedia.org/wiki/Enterprise_content_management*

Wikipedia (n.d.*b*), 'Hypertext transfer protocol'. Accessed: 08/25/2012.
URL: *http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol*

Zhu, W.-D., Lou, Y. Z., Meissler, J., Rathgeber, R., Richards, H., Saalfeld, J., Schmid, S. & Tachtevrenidis, K. (2012), *Customizing and Extending IBM Content Navigator*. IBM Redbook.

Zyp, K. (n.d.), 'Using Dojo Data'.
URL: *http://dojotoolkit.org/documentation/tutorials/1.6/dojo_data/*